Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Màster en Computació

Master Thesis

Design, Analysis and Implementation of New Variants of Kd-trees

Student: MARIA MERCÈ PONS CRESPO
Director: SALVADOR ROURA

Date: September 8, 2010

Acknowledgements

This master thesis would have not been possible without the constant help and support of my advisor Salvador Roura. I have learnt a lot from him, in particular from his enthusiasm and perseverance when working in interesting and challenging problems. I have enjoyed also very much our research discussions, that happened less often than I would have liked because of my job outside the University. I want to thank also Conrado Martínez for useful advice and encouraging me to do a thesis in this subject.

Finally, I thank all my family for their support, caring and love, specially to my mother, who always encourages me to pursue my goals. But I dedicate this thesis to my little nephew and niece, Marc and Olga, for just being so cute and delighting me with so many happy moments.

Contents

Chapter 1

Introduction

The representation of multidimensional data is a central issue in database design, as well as in many other fields, including computer graphics, computational geometry, pattern recognition, geographic information systems and others. Indeed, multidimensional points can represent locations, as well as more general records that arise in database management systems. For instance, consider an employee record that has attributes corresponding to the employee's name, address, sex, age, height and weight. Although the different dimensions have different data types (name and address are strings of characters; sex is a binary field; and age, height and weight are numbers), these records can be treated as points in a six-dimensional space.

We may see a database as a collection of records. Each record has several attributes, some of which are keys. The *associative retrieval problem* consists of answering queries with respect to a file of multidimensional records. Such an associative query requires the retrieval of those records in the file whose key attributes satisfy a certain condition. Examples of associative queries are intersection queries and nearest neighbor queries.

In order to facilitate the retrieval of records based on some conditions on its key attributes, it is usually helpful to assumed the existence of an ordering for its values. In the case of numeric keys, such an ordering is quite obvious. In the case of alphanumeric keys, the ordering is usually based on the alphabetic sequence of the characters making up the attribute value. Furthermore, certain queries, like nearest neighbor searches, require the existence of a distance function.

Several data structures for information retrieval systems support associative queries and offer different trade-offs of efficiency [Sam05]. Their space requirements, worst-case and expected-case performance on a range of operations, as well as the ease of their implementation, make them more

or less suitable for the dynamic maintenance of a file. The k-dimensional binary tree [Ben75] (or kd-tree, for short) is a convenient data structure because it supports a large set of operations with relatively simple algorithms, and offers reasonable compromises on time and space requirements. For this reason, we have taken this data structure as the basis for our work.

Chapter 2 recalls the *standard kd-tree* and describes some of the most important associative queries: full-defined search, orthogonal range, partial match, linear match, radius range and nearest neighbor. For each one of these queries, a pseudo-code algorithm for it is given, together with an example of execution.

The expected cost of a search, a partial match and a linear match in standard kd-trees is analyzed in Chapter 3. For the search and the partial match queries, the expected cost was previously known [FP86], but the analysis of the linear match is one of the contributions of this work.

Chapter 4 includes two already known variants of standard kd-trees: the *squarish kd-tree* of Chanzy, Devroye and Zamora-Cura [CDZc99] and the *relaxed kd-tree* of Duch, Estivill-Castro and Martínez [DECM98]. These two variants modify the insertion procedure of standard kd-trees, but all them share the same algorithms for associative queries.

In Chapter 5 we propose a new variant of kd-tree, the *median kd-tree*, which also share with the rest of variants of kd-trees the same algorithms for associative queries. Furthermore, we perform the corresponding theoretical analysis for the expected cost of a search and a partial match query.

In Chapter 6 we propose some hybrid variants that we obtain by combining standard kd-trees with squarish, relaxed and median kd-trees. We call the new variants *hybrid squarish*, *hybrid relaxed* and *hybrid median* kd-trees, respectively. The theoretical analysis of these variants for several operations is included in the same chapter.

Moreover, we have also implemented all the associative queries mentioned above, for the seven kd-tree variants of our interest. Chapter 7 presents the results of the experimental study that we have carried out. These results completely match with the theoretical results presented here, both those already in the literature and our new results.

Last but not least, another contribution of this work is an efficient library of generic metric and spatial data structures and algorithms, which we have implemented following the principles of the C++ Standard Template Library. Our library, that we have named *Spatial and Metric Template Library*, implements all the kd-tree variants and all the algorithms that we describe in Chapters 2 through 6. The components of the SMTL are robust, flexible and have been throughly tested, providing ready-to-use solutions to

common problems in the area, like nearest neighbor search or orthogonal range search. We have used SMTL library to run the experimental work presented in Chapter 7. We give instructions on its use, comments about its design and some implementation details in Chapter 8.

A final chapter is dedicated to present conclusions and a short discussion of possible future work.

Chapter 2

Standard kd-trees

A *kd-tree* is a well-known space-partitioning data structure for storing points of a k-dimensional space. This data structure is easy to understand and implement, can be conveniently updated and maintained, and it is useful for searches involving multidimensional keys. Moreover, it is appropriate for other queries like orthogonal range searches, partial match queries and nearest neighbor searches, among others.

In this chapter we present the *standard* kd-tree data structure. In the following chapters we will present several variants of kd-trees, which differ from the standard in the insertion procedure, in particular in the way to choose one of the k dimensions to split the search space. Nevertheless, exactly the same queries can be applied to all these kd-tree variants.

2.1 Definition of standard kd-trees

The kd-tree is a structure proposed by Bentley [Ben75] that generalizes the binary search tree for multiple dimensions. Consider a set of k-dimensional keys (say, arrays of size k starting at 0) that we want to store. Each node of the kd-tree has one of the keys and one discriminant associated to it. Every discriminant is an integer between 0 and $k-1$. Initially, the root represents the whole space. Let x be the key at the root and let i be its discriminant. Then, the space is partitioned in two regions with respect to $x[i]$: All the keys y with $y[i] < x[i]$ go into the left subtree, and all the keys with $y[i] \geq x[i]$ go into the right subtree. The same method of partitioning the space is recursively applied to all subtrees, until empty trees are reached. The discriminant at every node is chosen alternating the coordinates of each level, starting with 0: we use dimension 0 at the root, then dimension 1, then dimension 2, ..., then dimension $k-1$, then dimension 0 again, etc.

More formally, we have the following definition.

Definition 1 *A standard kd-tree for a set of k-dimensional keys is a binary tree in which:*

1. *Each node kas a key and an associated discriminant $i \in \{0, \ldots, k-1\}$.*
2. *For every node with key x and discriminant i, any key y in its left subtree satisfies $y[i] < x[i]$, and any key y in the right subtree satisfies $y[i] \geq x[i]$.*
3. *The root node has depth 0 and discriminant 0. All nodes at depth d have discriminant $d \bmod k$.*

Since the discriminants are assigned to nodes in a deterministic, simple way, the basic recursive algorithms for standard kd-trees could be implemented without explicitly storing the discriminants at the nodes. However, we explicitely include this field in the algorithms below, in order to present codes as general as possible, which can be directly used by other variants of kd-trees with no further modifications.

On the other hand, all the algorithms below are presented in a recursive way, and as intuitively as possible. Some details are avoided in the hope of not obscuring the code. By contrast, and for the sake of efficiency, the actual C++ implementations presented in Chapter 8 are iterative.

2.2 Inserting and Searching

The insert and search operations for kd-trees are similar to their counterparts for standard binary search trees, except that we have to use the appropriate coordinate at each level of the recursion. This information is given by the discriminant stored at each node of the kd-tree.

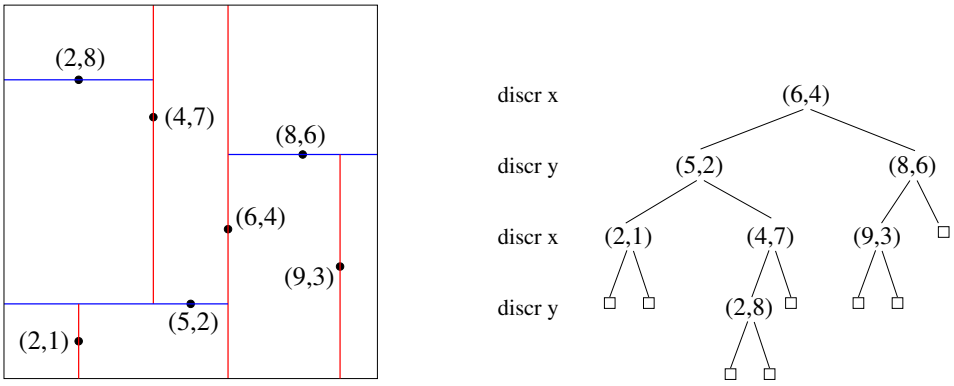
Algorithm 1 describes the procedure to insert an element with key k and value v into the kd-tree T. We suppose that the kd-tree does not already contain an element with key k before the insertion. If this were to be allowed, we should just add the condition $x = key$ to the code, and in this case update to v the old value associated to x.

The procedure InsertElement(x, v) inserts an element with key k and value v in the current leaf. For standard kd-trees, the discriminant should be chosen as the parent's discriminant plus 1 (modulo k). We do not include this detail, which depends on the kd-tree variant, in the algorithm.

Algorithm 1 Insertion in kd-trees.

```
procedure INSERT(T, x, v)
    if T = □ then INSERTELEMENT(x, v)
    key ← T.key; i ← T.discr
    if x[i] < key[i] then INSERT(T.left, x, v)
    else INSERT(T.right, x, v)
```

Figure 2.1 shows the kd-tree obtained after inserting $(6,4)$, $(5,2)$, $(4,7)$, $(8,6)$, $(2,1)$, $(9,3)$ and $(2,8)$ in this order into an initially empty kd-tree. In the figure, the total region is $[0,10] \times [0,10]$, assuming that we know that the inserted points will always fall into this square. The first cut, made by $(6,4)$, is vertical at 6. The second cut, made by $(5,2)$, is horizontal at 2 but only affects the $[0,6) \times [0,10]$ subregion. The third cut, made by $(4,7)$, is vertical at 4, and only affects the $[0,6) \times [2,10]$ subregion, etc.



(a) Partition of the plane

(b) 2d-tree structure

Figure 2.1: Inserting seven elements in a standard 2d-tree.

Note that every node represents both a point and a subregion of the plane, called bounding box. For instance, with the assumption above that all points fall into $[0,10] \times [0,10]$, the node with key $(4,7)$ represents this key but also the bounding box $[0,6) \times [2,10]$.

On the other hand, if every node kept information about the actual keys stored in its subtree, then the bounding box of that node would be $[2,4] \times [7,8]$, indeed the smallest rectangle that includes all keys currently stored in that subtree.

There is yet another possible scenario, in which we have no information about the points to come in the future, nor do we keep information about the points already inserted at every subtree. In that case, the bounding box of that node would be $(-\infty,6) \times [2,\infty)$. Note that this bounding box can be easily computed while descending from the root to $(4,7)$.

The algorithms presented in this work assume the last possibility, i.e., having and storing the minimum information. As a consequence, they already work in the two other settings. Moreover, all queries could be readily adapted if we wanted to take profit of the additional information to avoid exploring some useless subtrees.

Algorithm 2 includes the search procedure, which is similar to the insertion procedure. This code could be easily adapted to return the value associated to a given key x, if x is in T, or some dummy value otherwise.

Algorithm 2 Search in kd-trees.

```
procedure SEARCH(T, x)
    if T = □ then return  not found
    key ← T.key; i ← T.discr
    if x = key then return  found
    if x[i] < key[i] then return  SEARCH(T.left, x)
    else return  SEARCH(T.right, x)
```

As an example, if we search for $(2, 8)$ over the previous kd-tree, the key is found following the painted path.

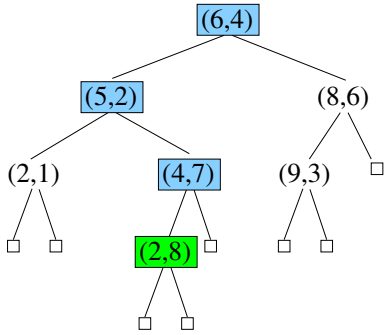


Figure 2.2: Search in a 2d-tree.

2.3 Orthogonal Range Searching

The orthogonal range search operation returns all the points within a given (hyper)rectangle. The query is specified by the lowermost and uppermost corners of the rectangle, $lowerBound$ and $upperBound$. The operation must return all keys x in the kd-tree inside the rectangle, that is, such that

$$lowerBound[i] \leq x[i] \leq upperBound[i]$$

for all $i, 0 \leq i < k$.

Algorithm 3 presents the orthogonal range procedure for kd-trees. The algorithm recursively visits all subtrees whose bounding box have a non-empty intersection with the query rectangle. Remember that we assume that we know and store the minimum information. So, for instance, the bounding box of the root is $(-\infty, \infty)^k$. Note that the bounding box of every node is not explicitely computed. Instead, the recursive calls just discard the subtrees whose implicit bounding box lie outside of the query rectangle.

In the algorithm, INSIDE($key, lowerBound, upperBound$) is a function that checks whether key is inside the rectangle defined by $lowerBound$ and $upperBound$.

Algorithm 3 Orthogonal Range in kd-trees.

```
procedure OrthogonalRange(T, lowerBound, upperBound, L)
    if T = □ then return
    key ← T.key; i ← T.discr
    if Inside(key, lowerBound, upperBound) then L ← L ∪ {key}
    if lowerBound[i] < x[i] then
        OrthogonalRange(T.left, lowerBound, upperBound, L)
    if upperBound[i] ≥ x[i] then
        OrthogonalRange(T.right, lowerBound, upperBound, L)
```

Figure 2.3 shows an example of orthogonal range query with a rectangle defined by $lowerBound = (1, 5)$ and $upperBound = (5, 9)$. In the 2d-tree, the nodes explored are painted green or blue, depending on whether they belong to the solution or not. In this example, the points falling within the rectangle are $(4, 7)$ and $(2, 8)$.

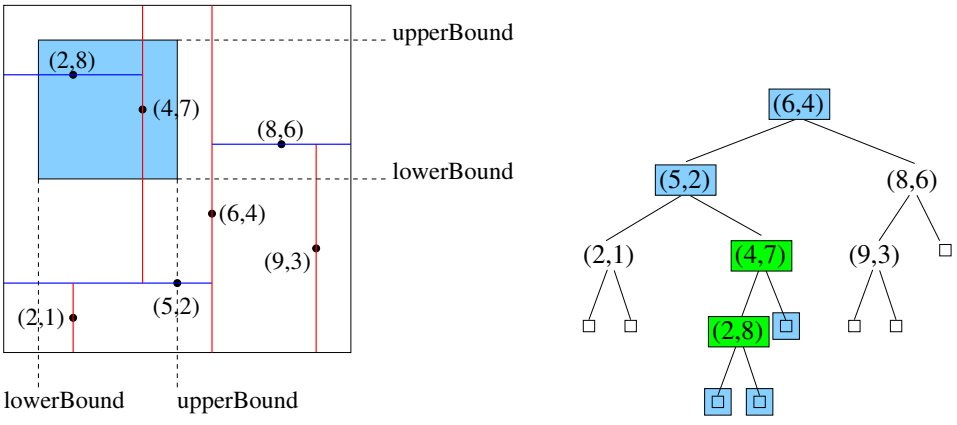


Figure 2.3: Orthogonal Range in a 2d-tree.

The steps that the orthogonal range algorithm follows in this example are described in the table below.

level 0	root (6,4)	VISIT
level 1	left subtree rooted at (5,2)	VISIT
	right subtree rooted at (8,6)	DISCARD
level 2	left subtree rooted at (2,1)	DISCARD
	right subtree rooted at (4,7)	VISIT
		found (4,7)
level 3	left subtree rooted at (2,8)	VISIT
		found (2,8)
	empty right subtree	VISIT
level 4	empty left subtree	VISIT
	empty right subtree	VISIT

2.4 Partial Match Query

A partial match query returns all keys that match a given k-dimensional query vector v with several wild cards, that is, with some unspecified dimensions. For instance, if we search for the keys that match $v = [a, *, *]$, where $*$ denotes a wild card, then we look for keys x such that $x[0] = a$, no matter which is the value of $x[1]$ or $x[2]$.

Algorithm 4 describes the procedure for a partial match, which is somehow similar to the orthogonal range algorithm. In fact, a partial match search is a particular (degenerated) case of orthogonal range query, where the range values for some dimensions (the specified ones) are reduced to just one point, while for the rest of dimensions the range is $(-\infty, \infty)$. In other words, if s is the number of specified values, then v represents a $(k - s)$-dimensional region of the space. For instance, a partial match with query vector $v = [a, *, *]$ corresponds to an ortogonal match defined by the points $(a, -\infty, -\infty)$ and (a, ∞, ∞).

In the algorithm below, MATCH(key, v) is a function that checks whether the current key matches the restrictions imposed by v. In that case, the key is added to the result list L.

At each node, the recursive calls depend on the vaule of $v[i]$, where i is the current discriminant. If it is specified, the algorithms is recursively called for the appropriate subtree. Otherwise, we recursively keep searching for keys into the two subtrees.

Algorithm 4 Partial Match in kd-trees.

```
procedure PartialMatch(T, v, L)
    if T = □ then return
    key ← T.key; i ← T.discr
    if Match(key, v) then L ← L ∪ {key}
    if v[i] = * then
        PartialMatch(T.left, v, L)
        PartialMatch(T.right, v, L)
    else
        if v[i] < key[i] then PartialMatch(T.left, v, L)
        else PartialMatch(T.right, v, L)
```

In Figure 2.4 we make a partial match query with $v = [8, *]$ to the 2d-tree already used before. Note that we search for all points whose x-coordinate is 8. All these points are situated over the dashed line, so the algorithm just returns the point $(8, 6)$.

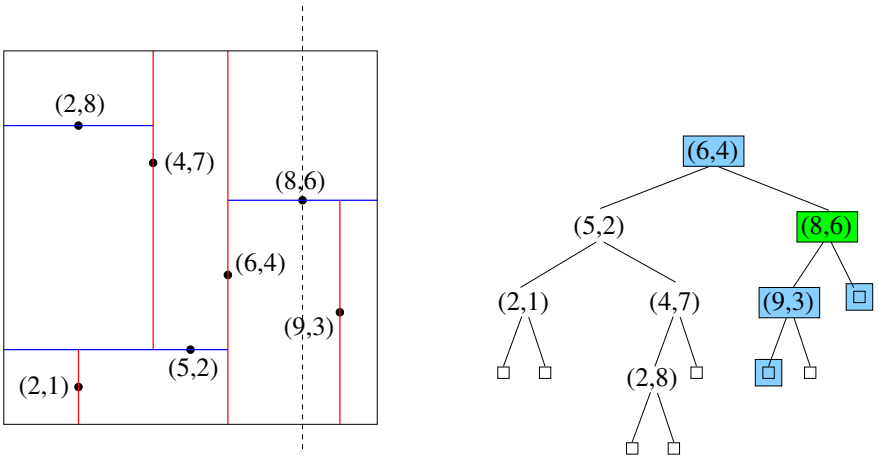


Figure 2.4: Partial Match in a 2d-tree.

The steps that the partial match query follows in this example are described in the table below.

level 0	root (6,4)	VISIT
level 1	left subtree rooted at (5,2) right subtree rooted at (8,6)	DISCARD VISIT found (8,6)
level 2	left subtree rooted at (9,3) empty right subtree	VISIT VISIT
level 3	empty left subtree empty right subtree	VISIT DISCARD

2.5 Linear Match Query

The linear match query operation returns all points located over a specific line. We assume that the line is defined by a point p and its slope s, although it could be defined by any equivalent way.

Note that a partial match is a particular case of linear match query. For instance, a partial match with query pattern $[x, *]$ is like a linear match with point $p = (x, y)$ for any y and slope $s = \infty$. Similarly, a partial match with query pattern $[*, y]$ is like a linear match with point $p = (x, y)$ for any x and slope $s = 0$.

Algorithm 5 includes the linear match procedure for kd-trees. It starts at the root of the kd-tree and recursively visits all subtrees whose bounding box is intersected by the input line.

In the algorithm, the function MATCH(key, p, s) uses simple geometry to check if key belongs to the given line.

The procedure COMPUTEBOUNDINGBOXES(...) returns the bounding boxes lBB and rBB for the left and for the right subtrees, respectively, given the current bounding box BB, the value $key[i]$ that will cut the bounding box and the discriminant of the current node. Note that this algorithm is the first that needs to explicitly compute the bounding box of every visited node. Otherwise, the subtrees that are useless for the query could not be detected and discarded. As we have said previously, the nodes do not need to keep additional information about the points stored at every subtree and the algorithm computes the boundings boxes while it moves down from the root.

In order to know if we need to explore a particular subtree, we use INTERSECTS(...), which tells if the input line intersects the corresponding bounding box. This function is a bit complicated to implement because several cases have to be taken into account.

Algorithm 5 Linear Match in kd-trees.

```
procedure LINEARMATCH(T, BB, p, s, L)
    if T = □ then return
    key ← T.key; i ← T.discr
    if MATCH(key, p, s) then L ← L ∪ {key}
    COMPUTEBOUNDINGBOXES(lBB, rBB, BB, key[i], i)
    if INTERSECTS(p, s, lBB) then LINEARMATCH(T.left, lBB, p, s, L)
    if INTERSECTS(p, s, rBB) then LINEARMATCH(T.right, rBB, p, s, L)
```

As an example, suppose that we run a linear match query over the previous 2d-tree, using as input parameters the point $(8, 8)$ and slope 2. The algorithm returns all the points located over the dashed line of Figure 2.5, that is, $(6, 4)$ and $(5, 2)$. Although, in this example, most nodes of the tree are visited, this does not have to be the case in general.

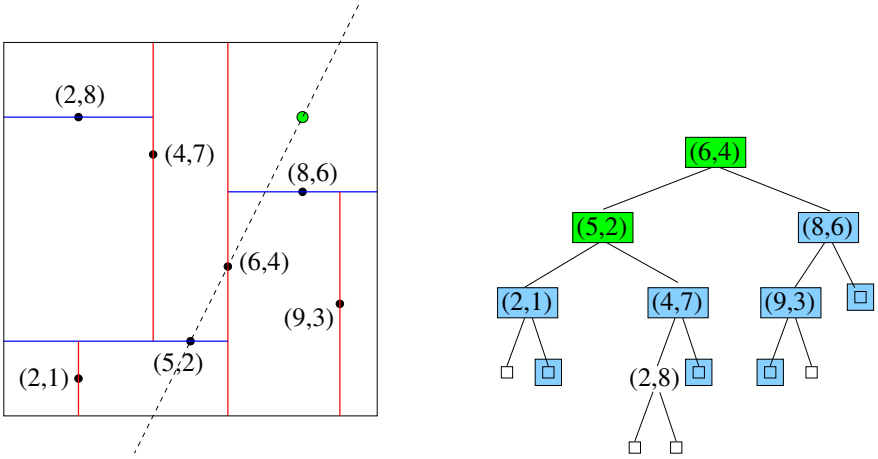


Figure 2.5: Linear Match in a 2d-tree.

The steps that the linear match query algorithm follows in this example are described in the table below.

level 0	root (6,4)	VISIT found (6,4)
level 1	left subtree rooted at (5,2) right subtree rooted at (8,6)	VISIT found (5,2) VISIT
level 2	left subtree rooted at (2,1) right subtree rooted at (4,7) left subtree rooted at (9,3) empty right subtree	VISIT VISIT VISIT VISIT
level 3	empty left subtree empty right subtree left subtree rooted at (2,8) empty right subtree empty left subtree empty right subtree	DISCARD VISIT DISCARD VISIT VISIT DISCARD

2.6 Radius Range Searching

"Distance" is a numerical description of how far apart two objects are. In mathematics, a distance function must satisfy the following conditions:

- it is positive: $d(x, y) \geq 0$;
- the distance from a point to itself is 0: $d(x, y) = 0$ if and only if $x = y$;
- it is symmetric: $d(x, y) = d(y, x)$;
- it satisfies the triangle inequality: $d(x, y) + d(y, z) \geq d(x, z)$.

Suppose that a certain distance function d is specified, together with a center point c and a radius r. The radius range searching operation returns all points p such that $d(c, p) \leq r$.

Algorithm 6 shows this procedure, which resembles the orthogonal range search. In the latter, we check if the current point is within a rectangle. In the former, we check if the current point lies inside the region defined by the distance constraint.

In Algorithm 6, the procedure ComputeBoundingBoxes(...) calculates the bounding boxes lBB and rBB for the left and for the right subtrees, respectively. These bounding boxes are then used by the function Intersects(BB, c, r), which tells if the bounding box BB intersects with the region that satisfies the distance constraints. If the intersection is non-empty, the subtree has to be explored.

Algorithm 6 Radius Range in kd-trees.

```
procedure RadiusRange(T, BB, c, r, L)
    if T = □ then return
    key ← T.key; i ← T.discr;
    if distance(key, c) ≤ r then L ← L ∪ {key}
    ComputeBoundingBoxes(lBB, rBB, BB, key[i], i)
    if Intersects(lBB, c, r) then
        RadiusRange(T.left, lBB, center, radius, L)
    if Intersects(rBB, c, r) then
        RadiusRange(T.right, rBB, center, radius, L)
```

Let us see an example of radius range search over the previous 2d-tree. Here, we use the Euclidean ("ordinary") distance, i.e., the length of the straight segment between two points. This distance is computed as usual,

$$distance(x, y) = \sqrt{\sum_{i=0}^{k-1} (x[i] - y[i])^2}.$$

With this definition, the points that satisfy $d(c, p) \leq r$ are those located within a ball with radius r centered at point p.

In Figure 2.6, suppose that we want to recover all the points that lie inside a ball with radius 1.5 centered at $(3, 7)$. The algorithm analyzes all that subtrees whose bounding boxes intersect with the input ball. In the example, the algorithm returns the points $(4, 7)$ and $(2, 8)$.

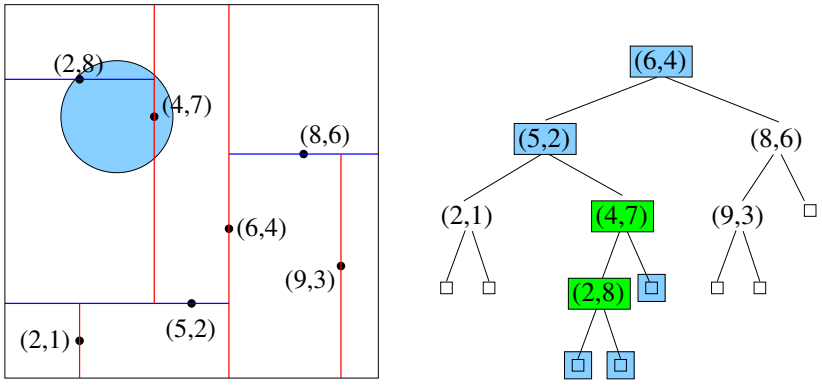


Figure 2.6: Radius Range in a 2d-tree using the Euclidean distance.

The steps that the radius range algorithm follows in this example are described in the table below.

level 0	root (6,4)	VISIT
level 1	left subtree rooted at (5,2)	VISIT
	right subtree rooted at (8,6)	DISCARD
level 2	left subtree rooted at (2,1)	DISCARD
	right subtree rooted at (4,7)	VISIT
		found (4,7)
level 3	left subtree rooted at (2,8)	VISIT
		found (2,8)
	empty right subtree	VISIT
level 4	empty left subtree	VISIT
	empty right subtree	VISIT

Suppose that we run the same example but using another distance: now we want to recover all points that have a Manhattan distance not larger than 1.5 from the center point $(3, 7)$. For this metric, the distance between two points is the sum of the absolute differences of their coordinates,

$$distance(x, y) = \sum_{i=0}^{k-1} |x_i - y_i|.$$

The points that satisfy the distance constraint are those located inside a rhombus whose diagonals mesure 3 units. Now, the algorithm analyzes every subtree whose bounding box intersects with the rhombus, and it checks the Manhattan distance instead of the Euclidean distance before inserting any point in the result list.

As Figure 2.7 shows, only the point $(4, 7)$ lies inside the rhombus. and the point $(2, 8)$ is no longer returned by the algorithm.

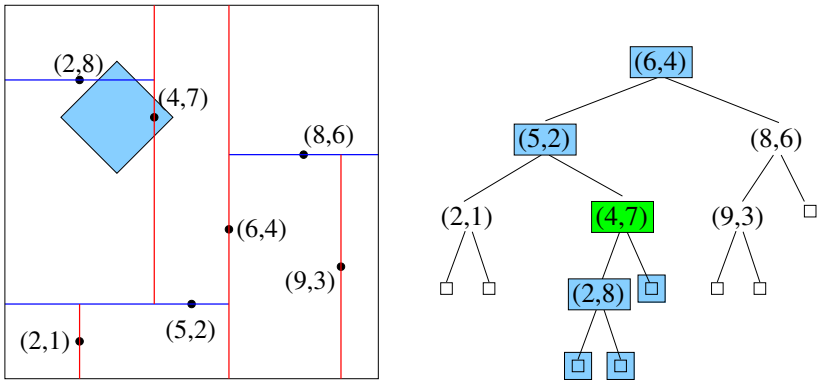


Figure 2.7: Radius Range in a 2d-tree using the Manhattan distance.

This time, the radius range query algorithm visits the nodes described in the table below.

level 0	root (6,4)	VISIT
level 1	left subtree rooted at (5,2) right subtree rooted at (8,6)	VISIT DISCARD
level 2	left subtree rooted at (2,1) right subtree rooted at (4,7)	DISCARD VISIT found (4,7)
level 3	left subtree rooted at (2,8) empty right subtree	VISIT VISIT
level 4	empty left subtree empty right subtree	VISIT VISIT

2.7 Nearest Neighbor Searching

The Nearest Neighbor Search returns the closest point to some given point according to a certain distance function.

We propose a very intuitive algorithm to solve this problem. Suppose that we run the nearest neighbor search with respect to a point c. When the algorithm explores some point of the kd-tree, it starts computing the distance between this point and c. If that is the minimum distance found until now, it stores this information, because the point is a candidate to be the nearest neighbor. Otherwise, the point is discarded. In any case, the algorithm computes the *potential* distance for their left and right subtrees, defined as the minimum possible distance between a point lying inside the bounding box of those subtrees and c. We call it "potential" because we do not know yet if such a point exists in the subtree.

We use a priority queue to store the information of the subtrees. Every item in the priority queue holds a pointer to the root of its corresponding subtree, its associated bounding box and the potential distance to c. The priority queue is sorted by potential distances, the minimum the better, and gives us the order to explore the subtrees to search for the nearest neighbor. At each iteration, the algorithm extracts the top of the priority queue. Then, it computes the real distance between the point at the root of the subtree and c. If this distance is smaller than the best found until now, we update the information consistently. The algorithm also computes the potential distances of the two subtrees. When a subtree has a potential distance lower than the minimum distance found so far, the information of the subtree is stored into the priority queue. Otherwise, the whole subtree is discarded.

The algorithm goes on until the priority queue gets empty, or until the top of the priority queue has an element with potential distance larger than the minimum real distance found so far. Then, we can safely state that the current candidate is in fact the closest point to c, because we have not found any point with smaller distance, and because it is impossible to find in the queue subtrees with points closer to c. Algorithm 7 shows this procedure.

In the algorithm, the procedure ComputeBoundingBoxes(...) returns the bounding boxes lBB and rBB for the left and the right subtrees, respectively. The function MinimumDistance(BB, c) returns the potential distance between any point located inside the bounding box BB and c.

Algorithm 7 Nearest Neighbor in kd-trees.

```
procedure NearestNeighbor(T, c)
    pq: PriorityQueue
    inf: tuple kdtree, bounding_box, potential_dist endtuple

    BB ← global bounding box of the k dimensional space
    nnKey ← undef; minDist ← ∞
    pq.push( inf(T, BB, 0) )
    while pq.size() > 0 and pq.top().potential_dist < minDist do
        T ← pq.top().kdtree; BB ← pq.top().bounding_box
        pq.pop()
        if T ≠ □ then
            key ← T.key; i ← T.discr;
            dist ← Distance(key, c)
            if dist < minDist then
                nnKey ← key; minDist ← dist
            ComputeBoundingBoxes(lBB, rBB, BB, key[i])
            pot_dist ← MinimumDistance(lBB, c)
            if pot_dist < dist then pq.push(inf(T.left, lBB, pot_dist));
            pot_dist ← MinimumDistance(rBB, c)
            if pot_dist < dist then pq.push(inf(T.right, rBB, pot_dist));
    return nnKey
```

Let us run the nearest neighbor search over the kd-tree used through all this chapter. As can be seen in Figure 2.8, the point closest to $(9, 8)$ with respect to the Euclidean distance is $(8, 6)$.

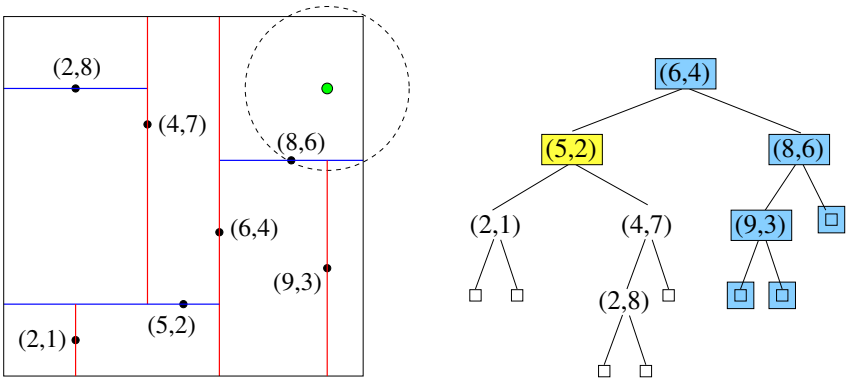


Figure 2.8: Nearest Neighbor in a 2d-tree.

When the query ends, the algorithm has explored all the nodes in blue, and the node in yellow is still stored in the priority queue. Note that the bounding box of the subtree rooted at $(5, 2)$, which is $(-\infty, 6) \times (-\infty, \infty)$, has potential distance 3 to the query point $(9, 8)$. Since this distance is larger than the minimum distance found by the algorithm, $\sqrt{5}$, it is not necessary to explore this subtree. On the other hand, the nodes in blue have a potential distance lower that the minimum distance found at the moment to analize them, so they had to be explored. For instance, the right subtree rooted at $(9, 3)$, despite being empty, has a bounding box $[9, \infty) \times (-\infty, 6)$, and therefore has potential distance 2 to the query point $(9, 8)$. Of course, the algorithm could be tuned to avoid storing empty trees into the queue, but we do not deal with those details here.

We end this chapter with a final remark. Although the algorithm presented here for the nearest neighbor search returns only the closest element to a given point, we have implemented an iterative version that allows us to get the second closest, the third closest and so on, in an incremental way. That is, we can get as many points as desired, sorted by increasing distance to the given point c. This incremental algorithm takes advantage of all previous iterations to get each new neighbor. In order to do that, it is necessary to store in the priority queue some additional information, but the idea of the algorithm is basically the same. We will explain the incremental version more carefully in Section 8.2.

Chapter 3

Analysis

The analysis included in this work refer to the cost of several algorithms on the average. To compute these average costs, we assume that the kd-trees are built from random data, say from a source of independent, uniformly distributed k-dimensional points chosen from $[0,1]^k$. We assume that the keys of the queries are also random.

In this chapter we analyze the cost of some algorithms, among them, the expected cost of a partial match and of a search in a standard kd-tree. These two results are well-known, but we include them here in order to get a more complete work, and also to introduce the mathematical techniques used henceforth.

Partial Match

Consider a standard 2d-tree built by inserting n points generated at random, that is, suppose that every point (x, y) is built by choosing x and y independently from a uniform distribution, say from $[0, 1]$ without loss of generality, and suppose also that the points are inserted into the kd-tree one by one, not using any balancing strategy.

Let X_n be the expected cost, measured as the number of visited nodes, of a partial search with only x defined in such a random kd-tree. Assume that the searched x is chosen uniformly at random from the range for x of the bounding box of the current subtree, being $[0,1]^2$ the bounding box of the root. This assumption will allow us to write a recurrence on just one variable, a recurrence that is amenable to analysis.

In the following equation, i denotes the x-rank (starting at 0) of the point at the root of the current subtree with n points. Since the kd-tree

is generated at random, each i has the same probability (namely, $1/n$), of being at the root. Moreover, it is not difficult to see that, conditioned to the fact that the i-th x is at the root, the probabilities of recursively searching into the left subtree or into the right subtree are respectively proportional to $i+1$ and $n-i$. Altogether,

$$X_n \;=\; 1 + \sum_{0\le i<n} \frac{1}{n}\left(\frac{i+1}{n+1}\cdot Y_i + \frac{n-i}{n+1}\cdot Y_{n-i-1}\right),$$

where Y_n denotes the expected cost of a partial search for a random x in an n-point random kd-tree that starts discriminating the points using first the coordinate y instead of x. Assuming $Y_0 = 0$, and by symmetry, we have

$$X_n \;=\; 1 + \sum_{0<i<n} \frac{2(i+1)}{n(n+1)}\cdot Y_i.$$

On the other hand, and by an argument similar to the one above, we get

$$Y_n \;=\; 1 + \sum_{0\le i<n} \frac{1}{n}\left(X_i + X_{n-i-1}\right) \;=\; 1 + \sum_{0<i<n} \frac{2}{n}\cdot X_i.$$

Therefore,

$$\begin{aligned} X_n &= 1 + \sum_{0<i<n} \frac{2(i+1)}{n(n+1)}\left(1 + \sum_{0<j<i} \frac{2}{i}\cdot X_j\right) \\ &= 2 + \sum_{0<j<n-1} \frac{4}{n(n+1)}\left(\sum_{j<i<n} \frac{(i+1)}{i}\right) X_j \\ &= 2 + \sum_{0<j<n-1} \frac{4}{n(n+1)}\left(n-j-1+H_{n-1}-H_j\right) X_j, \qquad (3.1) \end{aligned}$$

where H_n denotes as usual the n-th harmonic number, $H_n = \sum_{i=1}^{n} 1/i$.

If we denote w_j the weight of each of the X_j's in (3.1), in other words, $X_n = 2 + \sum_{j=0}^{n-1} w_j X_j$, then, for large n, the weights adapt to the shape function $w(z) = 4(1-z)$. Informally speaking, we have $w_j \sim w(j/n)/n$, with a small enough error approximation (see [Rou01]).

The solution to (3.1) is $X_n = \Theta(n^\alpha)$, where α is a positive number that is the unique solution to

$$1 \;=\; \int_0^1 w(z) z^\alpha dz \;=\; 4\int_0^1 (z^\alpha - z^{\alpha+1}) dz \;=\; 4\left(\frac{1}{\alpha+1} - \frac{1}{\alpha+2}\right),$$

or equivalently, $(\alpha+1)(\alpha+2) = 4$, which turns out to be $\alpha = (\sqrt{17}-3)/2$.

The analysis above could be done more informally, but perhaps more intuitively, as follows. Assume that n is very large. Then,

$$X_n \simeq 1 + \int_0^1 (x \cdot Y_{xn} + (1-x) \cdot Y_{n-xn})\, dx.$$

Note that the integral between 0 and 1 represents the continuous probability distribution of the coordinate x of the point at the root of the kd-tree. Conditioned to the fact that a fixed x is at the root, the probability to recursively search into the left subtree (that is, the probability that the randomly chosen value for the partial search is smaller than x) is precisely x. Additionally, the expected number of elements in the left subtree is approximately xn. An equivalent argument applies to the right subtree. Therefore, using symmetry, we have

$$X_n \simeq 1 + 2\int_0^1 x \cdot Y_{xn}\, dx. \tag{3.2}$$

Similarly, we can get

$$Y_n \simeq 1 + \int_0^1 (X_{yn} + X_{n-yn})\, dy = 1 + 2\int_0^1 X_{yn}\, dy.$$

Joining both approximations yields

$$\begin{aligned} X_n &\simeq 1 + 2\int_0^1 x \left(1 + 2\int_0^1 X_{yxn}\, dy\right) dx \\ &= 2 + 4\int_0^1 x \int_0^1 X_{yxn}\, dy\, dx. \end{aligned}$$

Let $z = yx$ in the inner integral. Then $dz = x\, dy$, and

$$\begin{aligned} X_n &\simeq 2 + 4\int_0^1 \int_0^x X_{zn}\, dz\, dx \\ &= 2 + 4\int_0^1 X_{zn} \left(\int_z^1 1\, dx\right) dz \\ &= 2 + 4\int_0^1 (1-z) X_{zn}\, dz. \end{aligned}$$

Under the hypothesis that $X_n \sim cn^\alpha$ for some constants $c > 0$ and $\alpha > 0$, for large n we can discard the term 2 above, and get

$$cn^\alpha \sim 4\int_0^1 (1-z) c(zn)^\alpha\, dz,$$

which implies

$$1 = 4\int_0^1 (1-z) z^\alpha\, dz,$$

whose solution, as we already know, is

$$\alpha \;=\; (\sqrt{17}-3)/2 \;\simeq\; 0.56155.$$

To conclude, the expected cost of a partial match operation in a standard kd-tree is

$$\Theta(n^{0.56155\ldots}). \tag{3.3}$$

Note that neither this analysis nor the previous one allows us to compute the constant factor c of the main term of X_n. The computation of c requires complete information of the algorithm even for small values of n (that is, it cannot be computed only through asymptotic information) and, moreover, involves using sophisticated mathematical techniques that are beyond the purpose of this work.

In what follows, and for the sake of brevity and clarity, for our analysis we will use the second (more intuitive) approach above, since it produces the same asymptotic results than the first (more rigorous) approach. Informally speaking, the fact that both approaches give the same result is precisely what was proved in [Rou01].

In a 2d-tree, the expected cost of a partial match is $\Theta(n^{0.56155\ldots})$, independently of the dimension (x or y) fixed in the query. Even so, the constant factor does depend on the dimension. Although our techniques do not allow us to compute the constants, we can at least compute the ratio of these constants using reasonable hypotheses.

Assume $X_n \simeq c_x \cdot n^\alpha$ and $Y_n \simeq c_y \cdot n^\alpha$, and plug both expressions into Equation 3.2. Then we have

$$c_x n^\alpha \;\sim\; 1 + 2\int_0^1 x \cdot c_y (xn)^\alpha dx \;=\; 1 + 2\,c_y\,n^\alpha \cdot \frac{1}{\alpha+2},$$

which implies

$$c_x \;=\; \frac{2c_y}{\alpha+2},$$

and using the value of α from (3.3), we get

$$c_y \;=\; \frac{(\sqrt{17}+1)\,c_x}{4} \;\simeq\; 1.28 c_x, \tag{3.4}$$

that is, the constant factor is around 28% larger when the fixed dimension is y instead of x.

It is not surprising that the number of elements visited during a partial match in a standard 2d-tree is larger when the y dimension is specified. Remember that x is always the discriminant used at the root. Therefore,

if the search fixes the x dimension, we discard a whole subtree in the first step. By contrast, if the search specifies the y dimension, we have to explore both subtrees, and we do not get rid of some subtrees until the next step.

The previous analysis was made for $k = 2$, with exactly one coordinate specified. Other results for larger k are already known. For instance, if we construct an off-line kd-tree, obtaining a perfectly balanced binary tree, the cost of a partial match in such a tree with n nodes is

$$\Theta(n^{1-s/k}), \tag{3.5}$$

where $s < k$ is the number of specified attributes in the query.

Similarly, the expected cost of a partial match in a random standard kd-tree is

$$P_n \quad = \quad \Theta(n^{1-s/k+\theta(s/k)}), \tag{3.6}$$

where $\theta(u)$ is a strictly positive function whose magnitude never exceeds the value 0.07 (see [FP86]).

Search

Let us consider the expected cost S_n of a completely specified random search in a random 2d-tree with n points. Now we have

$$\begin{aligned} S_n &\simeq 1 + \int_0^1 (x \cdot S_{xn} + (1-x) \cdot S_{n-xn})\, dx \\ &= 1 + 2\int_0^1 x \cdot S_{xn}\, dx. \end{aligned} \tag{3.7}$$

Note that, this time, the fact that the root of the current subtree discriminates using the x coordinate or the y coordinate makes no difference, so we can avoid an intermediate step and write S_n directly in terms of S_{xn}. Furthermore, this analysis applies for any dimension, not just $k = 2$. Under the hypothesis that $S_n \sim c \ln n$, we get

$$c \ln n \quad \sim \quad 1 + 2\int_0^1 xc \ln(xn)\, dx \quad = \quad 1 + 2c\int_0^1 (x \ln x + x \ln n)\, dx,$$

which turns out to be $c \ln n + 1 - c/2$. This implies $c = 2$, and

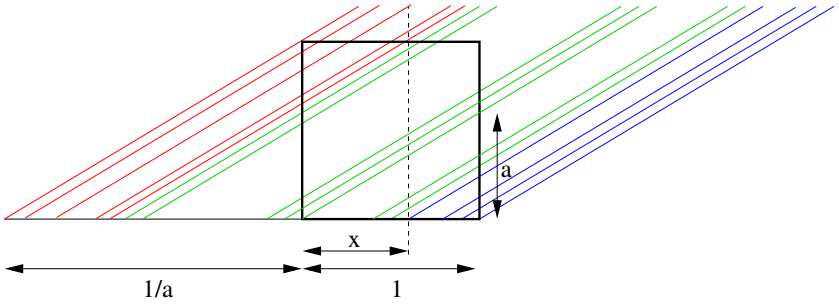
$$S_n \quad \sim \quad 2 \ln n \quad = \quad (2 \ln 2) \log_2 n \quad \simeq \quad 1.38629 \log_2 n. \tag{3.8}$$

Indeed, the expected cost of a random search in a random standard kd-tree with n points is the same as the expected cost of a random search in a random binary search tree with n keys, because the structure of both trees is identical, and so is the behavior of the respective search procedures.

Linear Match

Let us analyze the cost a linear match operation in a 2d-tree of size n. Specifically, we want to study how the slope of the line affects the expected cost of the operation.

Assume that the bounding box of the root is $[0,1]^2$. Fix a slope a. In our model, we will suppose that all the lines of slope a that cross the current bounding box are equally likely to be chosen for the linear match query. This scenario is shown in this figure:



Conditioned to the fact that in the first level the discriminant is x, the probability that a line with slope a cuts the left subtree (this corresponds to the red and green lines) is

$$\frac{\frac{1}{a}+x}{\frac{1}{a}+1} = \frac{1+ax}{1+a}\ .$$

In the following level of the tree, the discriminant is y, which in principle would imply a slope of $1/a$ in the recurrence. However, we need to scale the region to get a square again, so that the recursive subproblem is identical to the original one. Altogether, it is not difficult to see that the recursion applies for a line with slope $1/ax$. Therefore, and using symmetry with the green and blue lines, the expected cost $S_a(n)$ of a random linear match with slope a in a 2d-tree of size n is

$$\begin{aligned} S_a(n) &\simeq 1 + 2\int_0^1 \frac{1+ax}{1+a} S_{\frac{1}{ax}}(xn)\,dx \\ &= 1 + \frac{2}{1+a}\int_0^1 (1+ax) S_{\frac{1}{ax}}(xn)\,dx. \end{aligned} \tag{3.9}$$

Since a partial match is a particular case of linear match, it seems reasonable to assume $S_a(n) \simeq \beta(a)\, n^\alpha$, where α is the same exponent as in the cost of a partial match, that is, $\alpha = (\sqrt{17}-3)/2$, and the factor $\beta(a)$ depends on the slope a of the line. (For instance, we already know from

the analysis of a partial match that $\beta(0) \simeq 1.28\,\beta(\infty)$. Indeed, when the query line is horizontal ($a = 0$), this is like a partial match that specifies the y coordinate. When the query line is vertical ($a = \infty$), this is like a partial match that specifies the x coordinate.) Substituting $S_a(n) \simeq \beta(a)\,n^\alpha$ into (3.9), and simplifying, we get

$$\beta(a) \quad = \quad \frac{2}{1+a}\int_0^1 (1+ax)x^\alpha \beta\left(\frac{1}{ax}\right) dx. \tag{3.10}$$

To solve this differential equation, we perform several steps. To begin with, define

$$\xi(a) \quad = \quad (a+1)a^\alpha \beta(1/a). \tag{3.11}$$

Substituting into (3.10), and simplifying again, we get the simpler differential equation

$$\xi(a) \quad = \quad 2a^c \int_0^{1/a} \xi(z)\,dz, \tag{3.12}$$

where $c = 2\alpha + 2 = \sqrt{17} - 1$.

For the next step, let us assume for a moment that $\xi(a) = a^p$ for some $p \neq 1$. Then, we should have

$$a^p \quad = \quad 2a^c \int_0^{1/a} z^p\,dz \quad = \quad 2a^c\,\frac{(1/a)^{p+1}}{p+1} \quad = \quad \frac{2a^{c-p-1}}{p+1},$$

but unfortunately there is no solution for this equation. However, if we "iterate" again, we get

$$a^p \quad = \quad 2a^c \int_0^{1/a} \frac{2z^{c-p-1}}{p+1}\,dz \quad = \quad \frac{4a^c}{p+1}\cdot\frac{(1/a)^{c-p}}{c-p},$$

that is,

$$1 \quad = \quad \frac{4}{(p+1)(c-p)}.$$

The solutions for this equation are $p = c/2$ and $p = c/2 - 1$. Hence, the functions $a^{c/2}$ and $a^{c/2-1}$ are "fix points after two steps" of Equation (3.12). If we now assume that $\xi(a)$ is a linear combination of both functions,

$$\xi(a) \quad = \quad \delta_0\,a^{c/2} + \delta_1\,a^{c/2-1},$$

and we substitute into (3.12), we get

$$\xi(a) \quad = \quad 2a^c \int_0^{1/a} \left(\delta_0\,z^{c/2} + \delta_1\,z^{c/2-1}\right) dz \quad = \quad \frac{4\,\delta_0}{c+2}\,a^{c/2-1} + \frac{4\,\delta_1}{c}\,a^{c/2}.$$

From here we deduce $\delta_0 = 4\,\delta_1/c$, and $\delta_1 = 4\,\delta_0/(c+2)$. Since both equations are consistent, we have found a non-trivial solution to Equation (3.12)[1],

$$\xi(a) \;=\; \delta_0\, a^{c/2} + \frac{c}{4}\,\delta_0\, a^{c/2-1}.$$

Finally, if we use (3.11) backwards, we get the result

$$\beta(a) \;=\; \frac{1+(\sqrt{17}-1)a/4}{1+a}\,\delta_0,$$

for some unknown constant δ_0. Note that the existance of this unknown multiplying factor δ_0 is consistent with the results for partial match. Here, as there, this constant depends on non-asymptotic information, and thus it is not computable with our techniques.

On the other hand, we can check that indeed $\beta(0)/\beta(\infty) = (\sqrt{17}+1)/4$, as expected from Equation (3.4).

The plot below shows the function $\beta(\tan(x))$, where x is the angle of the query line. For the plot, we have arbitrarily set $\beta(0) = \delta_0 = 1$.

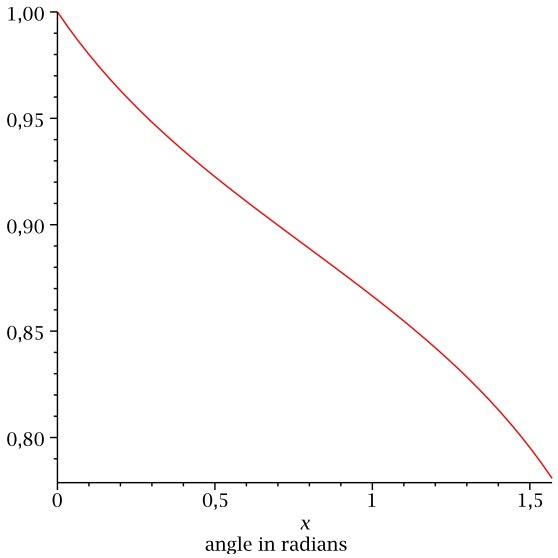


Figure 3.1: Function $\beta(\tan(a))$.

The results of the experiments carried out with the linear match query under this model are shown in Figure 7.11 on page 70. Our theoretical function completely matches the results of those experiments.

[1] Admittedly, we have not proven the uniqueness of the solution.

Orthogonal Range

In [DM02], Duch and Martínez presented the average-case analysis of the cost of orthogonal range searches for several multidimensional data structures. In that paper, it is proved that, for $k = 2$, the expected cost of a random centered range search of sides Δ_0 and Δ_1 in a kd-tree of size n, if the values of the Δ_i's are "small enough", is

$$\Delta_0 \, \Delta_1 \, n + \Theta(n^\alpha) + \Theta(\log n). \tag{3.13}$$

- The term $\Delta_0 \, \Delta_1 \, n$ is the expected number of reported points. This part of the expected cost is unavoidable since it must be paid in any case.

- The term $\Theta(n^\alpha)$ is the expected cost of a partial match, where α depends on the particular variant of kd-tree. This term is the dominant part of the overwork, and hence the term to look at when comparing the efficiency of the different variant of kd-trees.

  As we have previously seen, a partial match is a particular case of orthogonal range. Then, it makes sense this relation between both costs.

- The term $\Theta(\log n)$ is proportional to the expected cost of a search, and reflects the cost of moving down the tree from the root.

For larger values of k, there is a similar but more complex formula. For further details, see [DM02] and [CDZc99].

Chapter 4

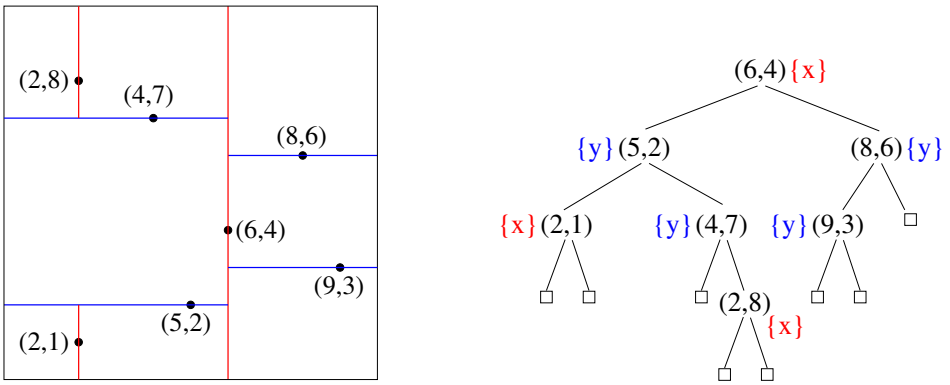
Squarish and relaxed kd-trees

4.1 Squarish kd-trees

Random standard kd-trees have not optimal performance for some operations like orthogonal range searching and partial match. Chanzy, Devroye and Zamora-Cura showed in [CDZc99] that this poor performance is due to the elongated character of most rectangles in the partitions of the planes defined by the kd-trees. In [DJZC00], they proposed a new kd-tree variant and analyzed its performance.

This variant consists in a modification of the way the discriminant is chosen at every node. When a rectangle is split by a newly inserted point, instead of alternating the discriminants, the longest side of the rectangle is cut. Therefore, the cut is always a $(k-1)$-dimensional hyperplane through the new point as usual, but now it is always perpendicular to the longest edge of the rectangle (if there is a tie, the discriminant can be chosen at random). As a result, these kd-trees have more squarish-looking regions, which is why this variant was named *squarish* kd-trees. Note that, compared to Definition 1 of standard kd-trees, this variant implies changes only in the third condition, so as to reflect the new method to choose the discriminant.

Figure 4.1 shows the squarish kd-tree obtained after inserting the same seven points of Figure 2.1 in the same order: $(6,4)$, $(5,2)$, $(4,7)$, $(8,6)$, $(2,1)$, $(9,3)$ and $(2,8)$. This time the plane is split in a different way. For instance, the point $(4,7)$ now splits the plane horizontally, because the vertical side of its bounding box is longer than the horizontal side. (In this example, we assume that it is known that the whole space is $[0,10]^2$, so that the bounding box of $(4,7)$ is $[0,6) \times [2,10]$.)



(a) Partition of the plane (b) 2d-tree structure

Figure 4.1: Inserting seven elements in a squarish 2d-tree.

Analysis

Some theoretical analysis for squarish kd-trees can be found in [DJZC00]. For instance, the expected cost of a search with n points is

$$S_n \simeq 1.38629 \log_2 n \tag{4.1}$$

for any $k \geq 2$, the same as that for standard kd-trees. The reason is that squarish kd-trees choose the discriminant by analyzing only the shape of the current region, choice which is is independent of the inserted point. Then, on the average the structure of the tree is the same than that of a random standard kd-tree. Indeed, for every squarish kd-tree with n points, the probability that the sizes of its left and right subtrees are i and $n-i-1$ are $1/n$ for every i between 0 and $n-1$.

Regarding a partial match, the expected cost for squarish kd-trees is $P_n = \Theta(n^{1-s/k})$, where $s < k$ is the number of defined coordinates. For the particular case of $k = 2$, we have

$$P_n = \Theta(\sqrt{n}\,), \tag{4.2}$$

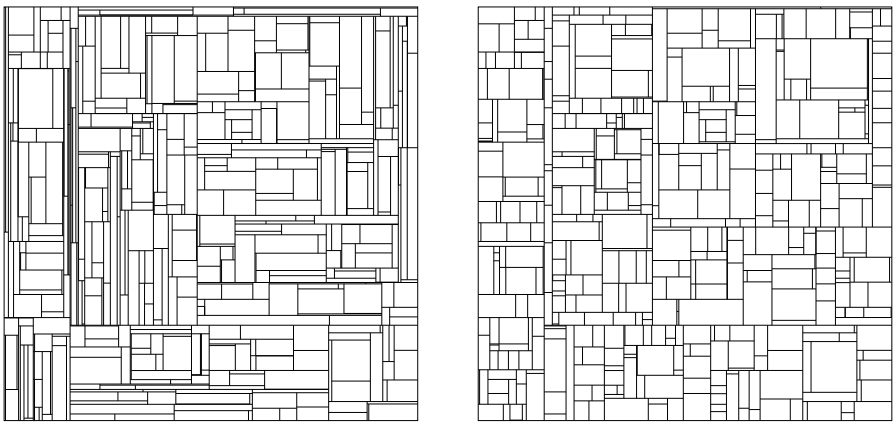
which is lower than the $\Theta(n^{0.56155\ldots})$ expected cost for standard kd-trees (see Equation 3.3). Note that the cost in the squarish case is proportional to that of an off-line kd-tree that splits the plane around the median at every node (see Equation 3.5). The good performance of this operation is due to the more squarish-looking regions.

This good performance of partial match also shows up in a very good performace of orthogonal range queries. For instance, when $k = 2$ the expected cost, as we have seen in Equation 3.13, is

$$\Delta_0\, \Delta_1\, n + \Theta(\sqrt{n}) + \Theta(\log n),$$

and the main term is unavoidable as it corresponds to the expected number of points returned by the query. Hence, the overwork $\Theta(\sqrt{n})$ is better than in the other variants of kd-trees.

Figure 4.2 shows how the plane is split for a standard and for a squarish 2d-tree when inserting the same 500 elements. We can see that the rectangles are less elongated in the squarish case, as expected. This explains its better performance for the partial match and for the orthogonal range search.



(a) standard kd-tree (b) squarish kd-tree

Figure 4.2: Splits in the plane for a standard and for a squarish kd-tree.

4.2 Relaxed kd-trees

Both standard and squarish kd-trees have strict restrictions on the choice of the discriminants. The former assigns them in a cyclic way, while the latter makes a choice according to the shape of the current region. Dealing with this restrictions force some update operations to be laborious, and better avoided if not made near the leaves.

Duch, Estivill-Castro and Martínez proposed in [DECM98] a new variant of kd-trees, called *relaxed* kd-trees. Their main advantage is their greater flexibility when performing update operations, because they do not have any restriction about the suitable discriminants at each node.

With this purpose, the construction of a relaxed kd-tree chooses the discriminants in a pure random way: when a new node is inserted, a random number from 0 to $k-1$ is picked as discriminant. Note that this choice is completely independent of previous decisions, of the structure of the tree, and of the inserted point. Compared to Definition 1 of standard kd-trees, relaxed kd-trees just drop the third condition.

Analysis

Running a search on a relaxed kd-tree of size n has expected cost

$$S_n \simeq 1.38629 \log_2 n \tag{4.3}$$

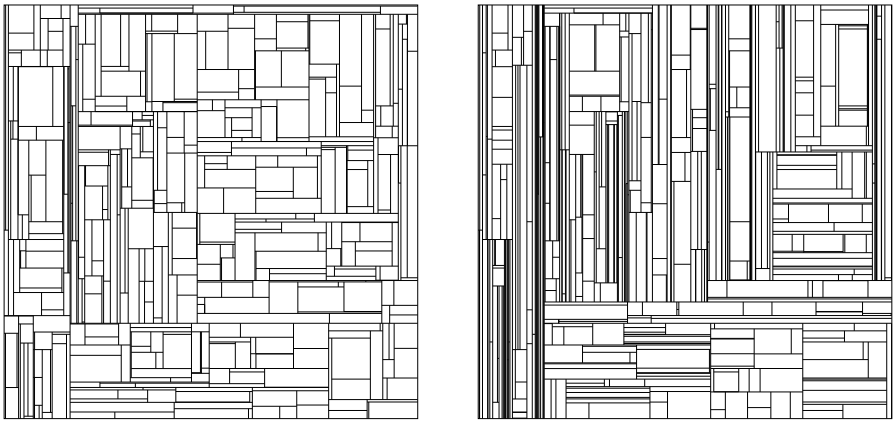
for any $k \geq 2$, like standard and squarish kd-trees. The reason is the same as before: the expected structure of the tree is identical to that of a random standard kd-tree.

The expected cost of a partial match with n points in a k-dimensional space, when there are $s < k$ specified coordinates, is $P_n = \Theta(n^{1-s/k+\theta(s/k)})$, that is, similar to that of standard kd-trees. However, here the upper bound for the function $\theta(u)$ is larger, around 0.12 (see [Duc04]). For instance, for the particular case $k = 2$, we have

$$P_n = \Theta(n^{0.61803\ldots}). \tag{4.4}$$

This also means that the overwork for orthogonal ranges is $\Theta(n^{0.61803\ldots})$, larger than the overwork for standard and squarish kd-trees.

Figure 4.3 shows the partitions of the plane for a standard and for a relaxed 2d-tree after inserting the same 500 elements. In a relaxed kd-tree, the regions are more elongated, which explains their larger cost for the partial match and for the orthogonal range.



(a) standard kd-tree (b) relaxed kd-tree

Figure 4.3: Splits in the plane for a standard and for a relaxed kd-tree.

Chapter 5

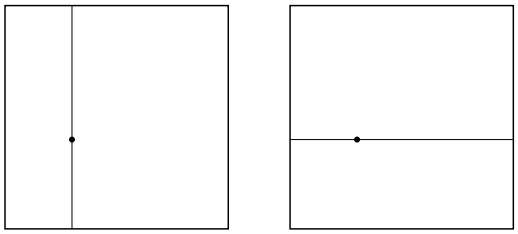
Median kd-trees

In some operations such as search or nearest neighbor, it is important to get a kd-tree as balanced as possible, because the expected cost is proportional to the height of the tree. But it is quite difficult to get a balanced kd-tree, unless it is built off-line. In this case, points can be inserted by selecting each time the point that splits around the median, obtaining an (almost) perfectly balanced kd-tree. However, when dynamic insertions and deletions are to be performed, a reorganization of the whole tree would be required. Thus, this alternative is not suitable unless updates occur rarely and most records in the file are known in advance. On the other hand, if the purpose is to construct a kd-tree from on-line insertions, balancing it is expensive, since kd-trees are sorted in multiple dimensions.

We propose a modification of the rule to assign discriminants that produces kd-trees more balanced than the standard kd-trees when the input data is independent and uniformly distributed. The modification is to always choose as a discriminant the dimension that "better" (in the sense defined below) cuts depending on the point that is inserted at this moment: when we insert a new point, all the coordinates are checked to know which one leaves two areas of size as similar as possible. Note that, since we are expecting randomly distributed points, both subtrees will be better balanced on the average than standard kd-trees. We call the new variant *median kd-tree*, because the coordinate chosen as discriminant is the one with the value that better approximates the expected median of the values that will be inserted in the current range.

For instance, suppose that we have an empty 2d-tree with points in the $[0, 1]$ range where we insert the $(0.3, 0.4)$ point. Let us see the behaviour of each of the kd-trees variants. A standard kd-tree chooses the 0-th dimension (x coordinate) because the point is at the root. A squarish kd-tree chooses a discriminant at random or makes a fixed decision because the starting area

is a square. A relaxed kd-tree always chooses a dimension at random, and it can be x or y with the same probability. However, in a median kd-tree we analyze the point. Choosing the x dimension produces two areas, one with 30% of the space and the other one with 70%. But choosing the y dimension produces two areas with 40% and 60% of the total area, respectively. In this case, we decide that it is better to cut along the y dimension. This scenario is shown in Figure 5.1, where we can see that the median kd-tree has made a good choice, at least with respect to the search operation.

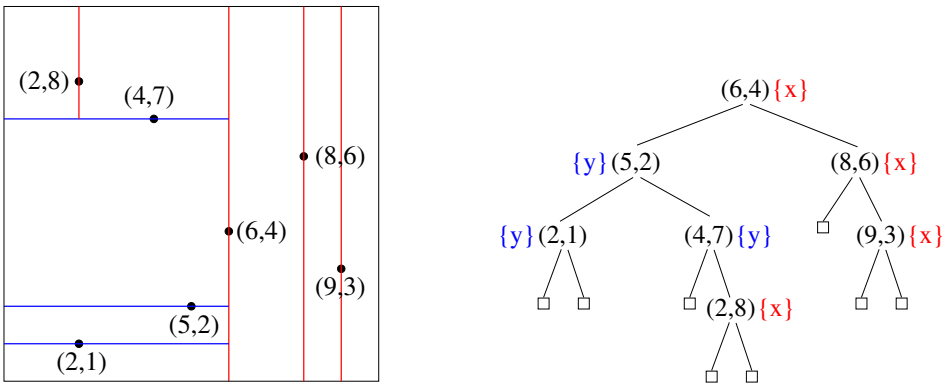


(a) Cutting along the x dimension

(b) Cutting along the y dimension

Figure 5.1: Inserting the $(0.3, 0.4)$ point at the root of a median kd-tree.

Figure 5.2 shows the median kd-tree obtained after inserting the same seven elements of Figures 2.1 and 4.1: (6,4), (5,2), (4,7), (8,6), (2,1), (9,3) and (2,8). The partition of the space that we get is different from the previous figures corresponding to the standard and the squarish kd-trees.



(a) Partition of the plane

(b) 2d-tree structure

Figure 5.2: Inserting seven elements in a median 2d-tree.

5.1 Analysis

Search

We begin the analysis of searches in median kd-trees in a 2-dimensional space. As in the analysis of standard kd-trees, we consider a kd-tree built by inserting n points generated at random. The only difference is that in this case the insertion method follows the variant proposed above.

The starting point in this computation is similar to the one for standard kd-trees (Eq. 3.7), but this time, we apply the symmetry described in Figure 5.3. Due to the fact that the points are randomly and uniformly distributed, we can state that their distribution in the eight sections is the same, so that the expected cost of searching in all the sections is equal. Additionally, it does not matter if the searching area is an elongated rectangle instead of a square, because we can scale the rectangle into a square without loss of generality.

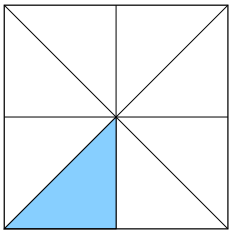


Figure 5.3: Symmetry in the searching area of a median kd-tree.

Then, what we do is to calculate the expected cost of a search in the shadowed area. After that, we multiply it by 8 to compute the cost in the whole area.

At each level, the algorithm discards one subtree and proceeds the search only into the appropriate subtree. Assume that the discriminant at the root is the x dimension. Conditioned to the fact that a fixed x is at the root, the probability to proceed the search into the left subtree is x, and the probability to proceed into the right subtree is $(1-x)$. The same argument could be applied if the discriminant at the root was y.

Therefore, we get

$$\begin{aligned} S_n &\simeq 1 + 8 \int_0^{1/2} \int_0^x \left(x\, S_{xn} + (1-x)\, S_{(1-x)n} \right) dy\, dx \\ &= 1 + 8 \int_0^{1/2} \left(x^2\, S_{xn} + x(1-x)\, S_{(1-x)n} \right) dx. \end{aligned} \tag{5.1}$$

Under the hypothesis that $S_n \sim c \ln n$, we have

$$\begin{aligned} c \ln n &\sim 1 + 8 \int_0^{1/2} \left(x^2 \, c \ln(xn) + x(1-x) \, c \ln((1-x)n) \right) dx \\ &= 1 + a + b, \end{aligned}$$

where

$$\begin{aligned} a &= 8c \ln n \int_0^{1/2} (x^2 + x(1-x)) dx = c \ln n, \\ b &= 8c \int_0^{1/2} \left(x^2 \, \ln x + x(1-x) \, \ln(1-x) \right) dx = 8c \left(\frac{\ln 2}{24} - \frac{5}{48} \right). \end{aligned}$$

Joining all these partial results, we get

$$c \ln n = 1 + c \ln n + 8c \left(\frac{\ln 2}{24} - \frac{5}{48} \right),$$

which implies

$$c = \frac{6}{5 - 2 \ln 2} \simeq 1.66035.$$

Hence, the expected cost of a random search in a median kd-tree for $k = 2$ is $S_n \sim 1.66035 \ln n$. Using base 2 logarithms,

$$S_n \sim \left(\frac{6 \ln 2}{5 - 2 \ln 2} \right) \log_2 n \simeq 1.15086 \log_2 n. \tag{5.2}$$

Contrary to our analysis of searches in standard kd-trees, the analysis here of searches in median kd-trees is only valid for $k = 2$. In fact, the expected cost of a search is always of the form $S_n \sim c_k \log_2 n$ but the constant c_k, which is $c_k \simeq 1.15086$ for $k = 2$, approaches the value $c_k = 1$ as the dimension grows, as we show now.

We carry out the analysis of searches in median kd-trees for several dimensions. First of all, we analyze the expected cost of a random search in a 3-dimensional space, and then we will generalize the result. In order to see the symmetry to apply in this scenario, Figure 5.4 shows the partitions that we make for each one of the 3 dimensions.

We divide the search area applying the red partitions and this division produces $2^3 = 8$ regions. Then, for each one of these regions there are three possible situations: split for the x, for the y or for the z coordinate. Therefore, we have 24 symmetric situations. Then, the expected cost of a search, when $k = 3$, can be represented by the recurrence

$$S_n \simeq 1 + 24 \int_0^{1/2} \int_0^x \int_0^x \left(x \, S_{xn} + (1-x) \, S_{(1-x)n} \right) dz \, dy \, dx,$$

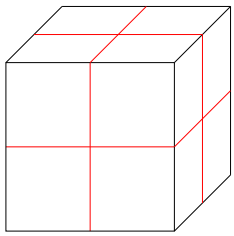


Figure 5.4: Partitions for the symmetry in the searching area of a median kd-tree for 3 dimensions.

and applying the hypothesis $S_n \sim c \ln n$, we get the equation

$$c \ln n = 1 + c \ln 2 + c \ln n - \frac{4}{3} c,$$

which implies that the constant c is

$$c = \frac{3}{4 - 3 \ln 2} \simeq 1.56205 \ln n.$$

Using base 2 logarithms,

$$S_n \sim \left(\frac{3 \ln 2}{4 - 3 \ln 2} \right) \log_2 n = 1.08273 \log_2 n.$$

Let us now generalize this result to k dimensions $(x_1, \ldots, x_k)$. If we apply partitions similar to those shown at Figure 5.4, we get 4 regions for 2 dimensions and 8 regions for 3 dimensions. In a 4-dimensional space, each of these partitions will be divided in two again, getting 16 regions. In general, the number of regions is 2^k. Since there are k dimensions to be used as discriminant, by symetry we have a multiplying factor of k. Then, the recurrence that expresses the expected cost for a random search in a k-dimensional space is

$$\begin{aligned}
S_n &\simeq 1 + k\, 2^k \int_0^{1/2} \int_0^{x_1} \ldots \int_0^{x_1} \left(x_1\, S_{x_1 n} + (1 - x_1)\, S_{(1-x_1)n} \right) dx_k \ldots dx_2\, dx_1 \\
&= 1 + k\, 2^k \int_0^{1/2} \left(x_1\, S_{x_1 n} + (1 - x_1)\, S_{(1-x_1)n} \right) dx_1 \int_0^{x_1} dx_2 \ldots \int_0^{x_1} dx_k \\
&= 1 + k\, 2^k \int_0^{1/2} \left(x_1\, S_{x_1 n} + (1 - x_1)\, S_{(1-x_1)n} \right) dx_1 \cdot x_1^{k-1} \\
&= 1 + k\, 2^k \int_0^{1/2} \left(x_1^k\, S_{x_1 n} + x_1^{k-1} (1 - x_1)\, S_{(1-x_1)n} \right) dx_1.
\end{aligned}$$

Under the hypothesis that $S_n \sim c_k \ln n$, we get

$$c_k \ln n \sim 1 + k\, 2^k \int_0^{1/2} \left(x_1^k\, c_k \ln(x_1 n) + x_1^{k-1} (1 - x_1) c_k \ln\left((1 - x_1) n\right) \right) dx_1$$

and

$$\begin{aligned} c_k &\sim \left(-k\,2^k \int_0^{1/2} \Big(x^k \ln(xn) + x^{k-1}(1-x)\ln\left((1-x)n\right) \Big)\, dx \right)^{-1} \\ &= \left[\frac{k}{2(k+1)} \left(\ln 2 - \frac{1}{k+1} \right) \right. \\ &+ \left. k\,2^k \sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} \left(\frac{-1}{(i+2)^2} + \frac{\ln 2}{(i+2)2^{i+2}} + \frac{1}{2^{i+2}(i+2)^2} \right) \right]^{-1}. \end{aligned}$$

If we compute the value of this constant c_k for several values of k, we get the following values:

$$\begin{aligned} c_2 &\simeq 1.15086, \\ c_3 &\simeq 1.08273, \\ c_4 &\simeq 1.05276, \\ c_5 &\simeq 1.03674, \\ c_{10} &\simeq 1.01118. \end{aligned}$$

As expected, the constant c_k tends to 1 when k increases. Although we have not proved it formally, it is quite evident that the expected cost for the search operation tends to

$$S_n \sim \log_2 n \tag{5.3}$$

as $k \to \infty$.

To sum up, the three kd-tree variants formerly presented (standard, squarish and relaxed) have an expected search cost about $1.38629\ \log_2 n$ for any dimension. However, a median kd-tree builds more balanced structures, and for this reason it has an expected search cost noticeably lower (around $1.15086\ \log_2 n$) in a 2-dimensional space, cost that tends to $1\ \log_2 n$ as the dimension grows.

Partial Match in two dimensions

In this section we analyze the expected cost of a partial match in a random median 2d-tree.

First of all we analyze the symmetry of the search area. Figure 5.5 shows which is the discriminant chosen for a point, depending on its location. All the points that are in the blue area use x as discriminant, because this is the coordinate that better cuts the area. For the same reason, the points in the green area use y as discriminant. There are eight equal areas, four of which use x as discriminant and the other four use y as discriminant.

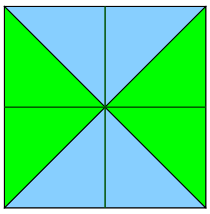


Figure 5.5: Symmetry in the search area for a partial match.

Consider that we want to calculate the expected cost when only x is specified. Assume that x is chosen at random. If the discriminant dimension is x, we only need to search into the appropriate subtree. If the searched value for the partial match is smaller than x we will proceed searching into the left subtree. The probability that this happens is x and the expected size of that subtree is xn. Using the same reasoning, with probability $1-x$ we proceed searching into the right subtree of expected size $(1-x)n$.

On the other hand, if the discriminant of the root is y, we need to search into the left and right subtrees with expected sizes xn and $(1-x)n$, respectively. Altogether,

$$\begin{aligned} P_n &\simeq 1 + 4\int_0^{1/2}\int_0^{x} \left(x\,P_{xn} + (1-x)\,P_{(1-x)n}\right) dy\,dx \\ &\quad + 4\int_0^{1/2}\int_0^{x} \left(P_{xn} + P_{(1-x)n}\right) dy\,dx \\ &= 1 + 4\int_0^{1/2}\int_0^{x} \left((x+1)\,P_{xn} + (2-x)\,P_{(1-x)n}\right) dy\,dx \\ &= 1 + 4\int_0^{1/2} \left(\left(x^2+x\right)\,P_{xn} + \left(2x-x^2\right)\,P_{(1-x)n}\right) dx. \end{aligned}$$

Under the hypothesis that $P_n \sim c\, n^\alpha$, for some constants $c > 0$ and $\alpha > 0$,

$$\begin{aligned} c\, n^\alpha &\sim 1 + 4\, c \int_0^{1/2} \Big((x^2 + x)(xn)^\alpha + (2x - x^2)\left((1-x)n\right)^\alpha \Big) dx \\ &= 1 + 4\, c\, n^\alpha (a + b), \end{aligned}$$

where

$$\begin{aligned} a &= \int_0^{1/2} \left(x^2 + x\right) x^\alpha dx = \frac{1}{(\alpha+3)\, 2^{\alpha+3}} + \frac{1}{(\alpha+2)\, 2^{\alpha+2}}, \\ b &= \int_0^{1/2} \left(2x - x^2\right) (1-x)^\alpha dx = \frac{1}{\alpha+1} - \frac{1}{\alpha+3} + \frac{1}{2^{\alpha+3}} - \frac{1}{\alpha+1}. \end{aligned}$$

This yields

$$1 = 2^{-\alpha} \left(\frac{1}{\alpha+3} + \frac{1}{\alpha+2} - \frac{2}{\alpha+1} \right) + \frac{4}{\alpha+1} - \frac{4}{\alpha+3},$$

whose solution is $\alpha \simeq 0.60196...$ Therefore, the expected cost for a partial match in a median kd-tree is

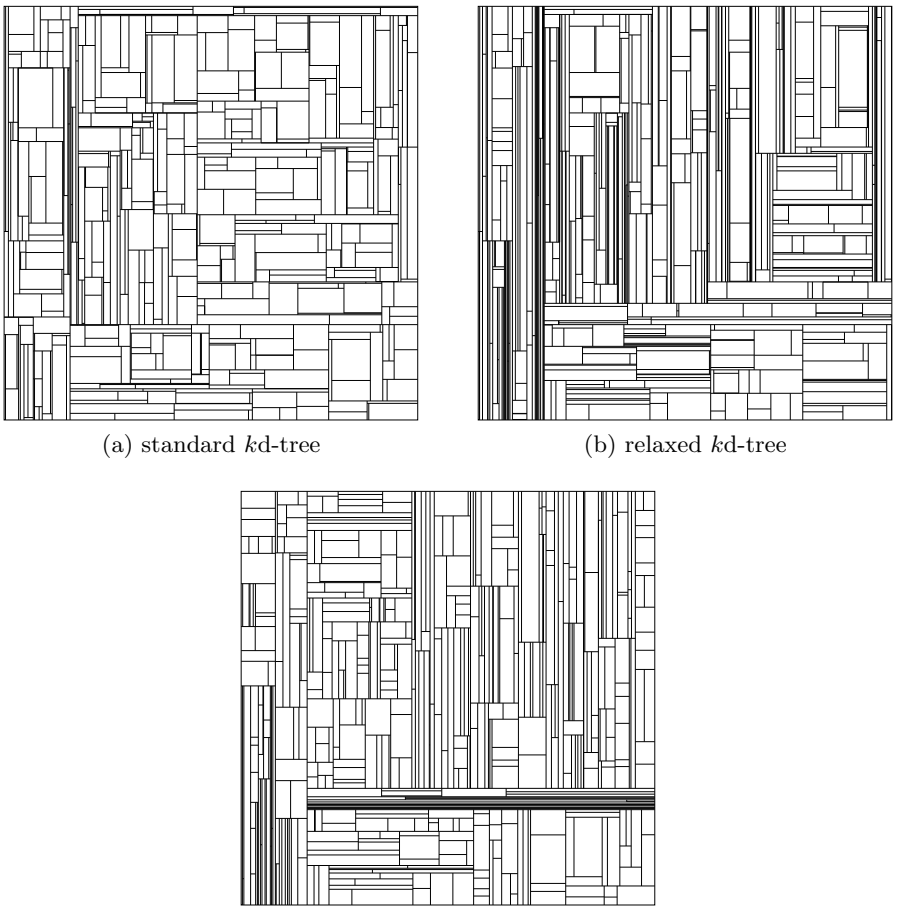
$$P_n = \Theta(n^{0.60196\ldots}). \tag{5.4}$$

Figure 5.6 shows the partitions of the plane for a standard, relaxed and median kd-tree after inserting the same 500 elements. We can observe that standard kd-trees have less elongated regions and thus a more efficient partial match operation (cost $\Theta(n^{0.56155\ldots})$, Eq. 3.3). Regarding the median and the relaxed kd-trees, the elongation of the partitions is similar, which corresponds to similar partial match costs: $\Theta(n^{0.60196\ldots})$ and $\Theta(n^{0.61803\ldots})$ (Eq. 4.4), respectively.

Partial Match in three dimensions

We now analyze the expected cost for a partial match in a median 3d-tree In this case, we have only two cases: one specified dimension or two dimensions specified.

Let us calculate the expected cost when only one dimension is specified. Similarly to the previous argument, if the discriminant coincides with the defined dimension, which in this case happens with probability 1/3, we will recursively proceed searching into the appropriate subtree. Otherwise (with probability 2/3), we will proceed searching into both subtrees. Here we use the same symmetry scenario than in the search operation, where the search



(a) standard kd-tree (b) relaxed kd-tree

(c) median kd-tree

Figure 5.6: Splits in the plane for a standard, relaxed and median kd-tree.

area is divided into 24 regions. Then,

$$\begin{aligned} P_n &\simeq 1 + \frac{1}{3}\,24 \int_0^{1/2} \int_0^x \int_0^x \left(x\,P_{xn} + (1-x)\,P_{(1-x)n}\right) dz\,dy\,dx \\ &\quad + \frac{2}{3}\,24 \int_0^{1/2} \int_0^x \int_0^x \left(P_{xn} + P_{(1-x)n}\right) dz\,dy\,dx. \end{aligned}$$

Applying the hypothesis $P_n \simeq c\,n^\alpha$ and assuming $\alpha > 0$, we get

$$P_n = \Theta(n^{0.74387\ldots}). \tag{5.5}$$

The opposite situation, when all the dimensions are specified except one, differs from the previous analysis in the probabilities to apply. In this case, if we have two specified dimensions over three, the probability that the

discriminant at the root coincides with a defined dimension is 2/3. In this case, the search proceeds into one subtree. And the probability to search into both subtrees, which happens when the discriminant does not match with a specified dimension, is 1/3. Therefore,

$$\begin{aligned} P_n &\simeq 1 + \frac{2}{3}\,24 \int_0^{1/2} \int_0^x \int_0^x \left(x\,P_{xn} + (1-x)\,P_{(1-x)n}\right) dz\,dy\,dx \\ &\quad + \frac{1}{3}\,24 \int_0^{1/2} \int_0^x \int_0^x \left(P_{xn} + P_{(1-x)n}\right) dz\,dy\,dx. \end{aligned}$$

Under the same hypothesis $P_n \simeq c\,n^\alpha$, we get

$$P_n = \Theta(n^{0.42756\ldots}). \tag{5.6}$$

Comparing both results, we can observe that the parameter α depends on the ratio of the dimensions specified in the query. Not surprisingly, this value decreases as more dimensions are specified. The expected cost of a partial match for larger k can be calculated by following the same steps presented here, although the computations get more cumbersome as k grows.

Chapter 6

Hybrid kd-tree variants

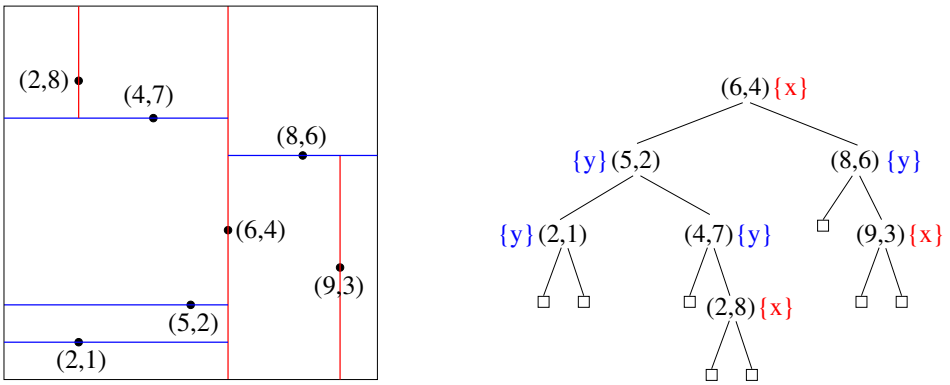
In previous chapters we have presented already existing kd-tree variants such as standard, squarish and relaxed kd-trees. We have also proposed and analyzed a new variant, the median kd-tree.

By analyzing the expected costs for several operations, we have seen that median kd-tree is the most efficient variant for the search operation, but it is less efficient than standard and squarish kd-trees for the partial match. Here, we propose several hybrid data-structure that combine standard kd-trees with other variants. We start with median kd-trees.

6.1 Hybrid median kd-tree

As with the other variants that we will present later, a hybrid median kd-tree modifies the way the discriminant is chosen. Consider that we want to insert some random points in a 2-dimensional hybrid median kd-tree. At the root, we choose as discriminant the one that "better" cuts the search area, depending on the point that it is inserted at this moment. In other words, at the 0-th level a hybrid median kd-tree works exactly as a median kd-tree. By contrast, when we move down to the next level, we alternate the discriminant, like standard kd-trees do. That is, if in the previous level the x coordinate was chosen, now we chose the y coordinate; and if in the previous level the y coordinate was chosen, now we chose the x coordinate. Once both coordinates have been used, we start again by choosing the discriminants as a median kd-tree.

Figure 6.1 shows the hybrid 2d-tree obtained after inserting $(6, 4)$, $(5, 2)$, $(4, 7)$, $(8, 6)$, $(2, 1)$, $(9, 3)$ and $(2, 8)$ in this order. Note that these points, the same used in the previous examples, split the plane differently.



(a) Partition of the plane

(b) 2d-tree structure

Figure 6.1: Inserting seven elements in a hybrid median 2d-tree.

Let us now generalize this result. Suppose that we want to insert some random points in a hybrid median kd-tree of k dimensions. Like in the 2-dimensional case, at the root level we choose the discriminant that "better" cuts the whole space. At the next level, we choose the coordinate that "better" cuts the region among the remaining $(k-1)$ coordinates, and so on, until only one coordinate remains unused. This happens at the $(k-1)$-th level. In this case, there are not alternatives and the procedure has to choose the remaining coordinate.

To summarize: a hybrid median kd-tree chooses the discriminants by taking into account the points inserted at every moment, and using the same cutting criterium as median kd-trees, but it ensures at the same time that every k levels all the coordinates are used exactly once.

6.1.1 Analysis

In order to analyze the expected cost for the hybrid kd-tree variants, we need to recover the recurrences of the previous sections. Since we will only analyze these hybrid kd-trees in 2-dimensional spaces, all the analysis have two "steps". In the hybrid median variant case, we use the median recurrences in the first step, and the standard recurrences in the second step.

Search

In this section we carry out the analysis of the hybrid median kd-tree for $k = 2$. We start analyzing the search operation. As we have explained in the definition of this new kd-tree variant, a hybrid median kd-tree works as a median kd-tree at the root, and it works as a standard at the following

level. Remember the recurrence that represents the cost for a search in both kd-tree variants (Eq. 5.1 and 3.7).

Consider the expected cost S_n of a completely specified random search in a hybrid median 2d-tree with n points. Let $S_n^{(1)}$ be the expected cost of a search in a hybrid median in the first step, and let $S_n^{(2)}$ be the cost of a search in the second step. Note that these recurrences are linked, so that one depends on the other. Therefore, we get

$$\begin{aligned} S_n^{(1)} &\simeq 1 + 8\int_0^{1/2}\int_0^x \left(x \cdot S_{xn}^{(2)} + (1-x)\cdot S_{(1-x)n}^{(2)}\right) dy\, dx. \\ S_n^{(2)} &\simeq 1 + 2\int_0^1 z \cdot S_{zn}^{(1)}\, dz. \end{aligned}$$

Joining both recurrences,

$$\begin{aligned} S_n &\simeq 1 + 8\int_0^{1/2}\int_0^x \left(x\cdot(1+2\int_0^1 z\cdot S_{zxn}\, dz)\right) dy\, dx \\ &+ 8\int_0^{1/2}\int_0^x \left((1-x)\cdot(1+2\int_0^1 z\cdot S_{z(1-x)n}\, dz)\right) dy\, dx \\ &= 1 + a + b, \end{aligned}$$

where

$$\begin{aligned} a &= \frac{1}{3} + 8\int_0^{1/2} wS_{wn}\, dw - 16\int_0^{1/2} w^2 S_{wn}\, dw \\ b &= \frac{2}{3} - 16\int_0^{1/2} wS_{wn}\left(\ln\frac{1}{2} + \frac{1}{2}\right) dw - 16\int_{1/2}^1 wS_{wn}(\ln w + 1 - w)\, dw. \end{aligned}$$

Combining all these results, and under the hypothesis that $S_n \sim c\ln n$, we get

$$c\ln n \sim 2 + \frac{1}{3}c\ln 2 - \frac{4}{3}c + c\ln n,$$

which implies

$$c = \frac{6}{4-\ln 2} \simeq 1.81441.$$

Using base 2 logarithms,

$$S_n \sim \left(\frac{6\ln 2}{4-\ln 2}\right)\log_2 n \simeq 1.25766\log_2 n. \tag{6.1}$$

As in the median kd-tree, this analysis is only valid in a 2-dimensional space, and the constant factor c decreases when the dimension grows. Although we have not formally proved it, the limit for this constant seems to be 1 too, but the convergence to the limit is slower in the hybrid median variant.

Partial Match

Let us analyze the expected cost of a partial match for a hybrid median kd-tree in a 2-dimensional space. As in the previous analysis, we need to have in mind the recurrences for the expected cost for median kd-tree and for standard kd-trees.

Suppose that the x coordinate is specified in the query. In the first step, we apply the recurrence for the median kd-tree. Then, if x is the discriminant, the algorithm visits only the appropriate subtree: the left subtree with probability x and the right subtree with probability $1-x$. If the discriminant is y, it visits both subtrees.

In the next step, the recurrence for the standard kd-tree is called. If the discriminant matched with the specified coordinate in the previous level, now it does not match in this level. That is, if in the previous level the algorithm visited only one subtree, now it has to visit both. And if previously it visited both subtrees, now it has to visit only one. Then, the global recurrence for the partial match operation is

$$\begin{aligned} P_n &\simeq 1 + 4\int_0^{1/2}\int_0^x \left(x \cdot Y_{xn} + (1-x)\cdot Y_{(1-x)n}\right) dy\, dx \\ &+ 4\int_0^{1/2}\int_0^x \left(X_{xn} + X_{(1-x)n}\right) dy\, dx, \end{aligned}$$

where X_n and Y_n are the recurrences for a standard partial match that applies in the second step. These recurrences are of the form

$$\begin{aligned} X_n &\simeq 1 + 2\int_0^1 z \cdot P_{zn}\, dz, \\ Y_n &\simeq 1 + 2\int_0^1 P_{zn}\, dz. \end{aligned}$$

Combining all these equations, and under the hypothesis that $P_n \sim c\, n^\alpha$, for some $\alpha > 0$,

$$\begin{aligned} P_n &\simeq 1 + 4\int_0^{1/2} x^2 + 2x^2 \int_0^1 P_{zxn}\, dz\, dx \\ &+ 4\int_0^{1/2} (x - x^2)(1 + 2\int_0^1 P_{z(1-x)n}\, dz\, dx \\ &+ 4\int_0^{1/2} x + 2x \int_0^1 z \cdot P_{zxn}\, dz\, dx \\ &+ 4\int_0^{1/2} x + 2x \int_0^1 z \cdot P_{z(1-x)n}\, dz\, dx \\ &= 1 + a + b + c + d, \end{aligned}$$

where

$$\begin{aligned} a &= \frac{1}{6} + c\,n^{\alpha}\,\frac{2^{-\alpha}}{(\alpha+1)(\alpha+3)}, \\ b &= \frac{1}{3} + c\,n^{\alpha}\,\frac{24-\alpha 3\cdot 2^{-\alpha}-12\cdot 2^{-\alpha}}{(\alpha+1)(\alpha+2)(\alpha+3)}, \\ c &= \frac{1}{2} + c\,n^{\alpha}\,\frac{2\cdot 2^{-\alpha}}{(\alpha+2)^2}, \\ d &= \frac{1}{2} + c\,n^{\alpha}\left(-\frac{2\cdot 2^{-\alpha}}{\alpha+1} - \frac{2\cdot 2^{-\alpha}}{(\alpha+1)(\alpha+2)} + \frac{8}{(\alpha+1)(\alpha+2)}\right). \end{aligned}$$

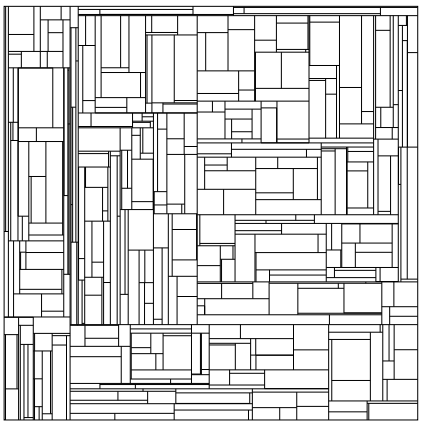
This yields that α is the unique positive real solution of

$$1 = \frac{8\alpha\cdot 2^{\alpha} - 3\alpha - 8 + 20\cdot 2^{\alpha}}{2^{\alpha-1}(\alpha+3)(\alpha+1)(\alpha+2)^2},$$

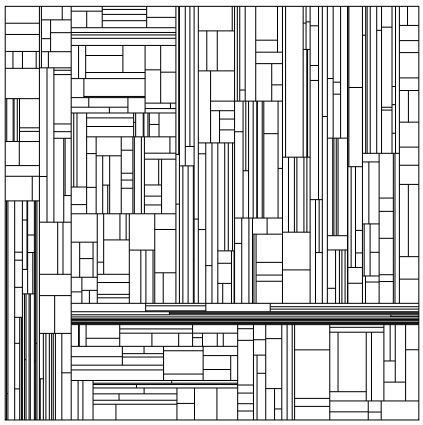
which is $\alpha \approx 0.54595$. Therefore, the expected cost for a partial match in a hybrid median kd-tree is, for $k = 2$,

$$P_n = \Theta(n^{0.54595\ldots}). \tag{6.2}$$

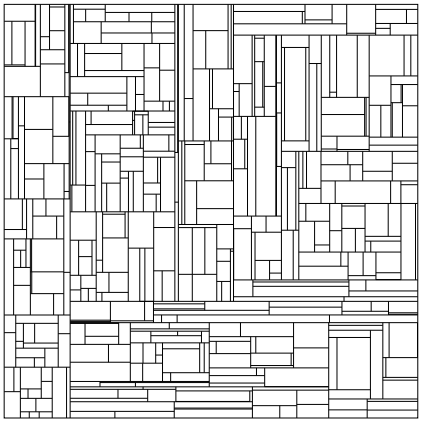
Figure 6.2 shows the partitions of the plane for a median kd-tree and a for a hybrid median kd-tree after inserting the same 500 elements. We can observe that the hybrid median variant produces more squarish regions, and for this reason it has a more efficient partial match operation: $\Theta(n^{0.54595\ldots})$ versus $\Theta(n^{0.60196\ldots})$. Note also that the cost of a partial match in hybrid kd-trees improves that of standard kd-trees: $\Theta(n^{0.56155\ldots})$.



(a) standard kd-tree



(b) median kd-tree



(c) hybrid median kd-tree

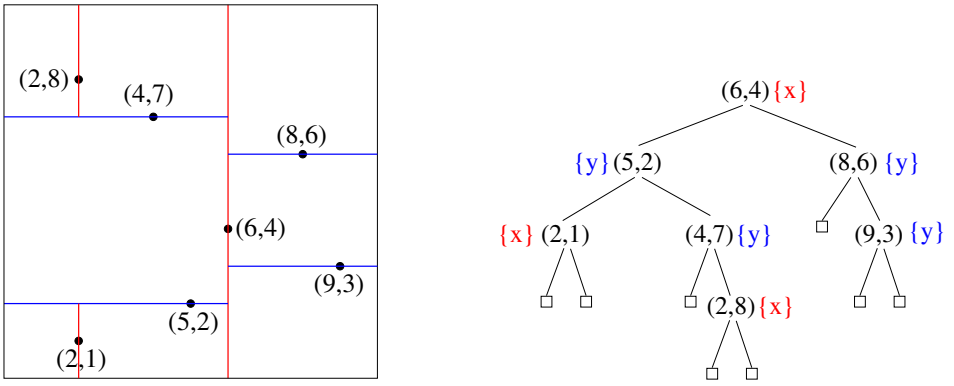
Figure 6.2: Partitions of the plane for a standard, median and hybrid median kd-tree.

6.2 Hybrid squarish kd-tree

The hybrid squarish kd-tree is similar to the hybrid median. In this case, the variant works as a squarish kd-tree at the first step, and it alternates the discriminant in the following step, as a standard kd-tree does. Considering that the root is at level 0, a hybrid squarish 2d-tree works as a squarish at the even levels, and it works as a standard kd-tree at the odd levels.

For k dimensions, we choose the first discriminant following the squarish kd-tree rules. In the following level, we choose the discriminant applying the squarish rules over the $k-1$ remaining discriminants, and so on, until we have used all the k discriminants in the k first levels.

Figure 6.3 shows the kd-tree obtained after inserting $(6,4)$, $(5,2)$, $(4,7)$, $(8,6)$, $(2,1)$, $(9,3)$ and $(2,8)$ points in this order, the same points used in the previous examples. In this case, we get the same partition of the plane that in the basic squarish kd-tree, but the partitions are different in general.



(a) Partition of the plane

(b) 2d-tree structure

Figure 6.3: Inserting seven elements in a hybrid squarish 2d-tree.

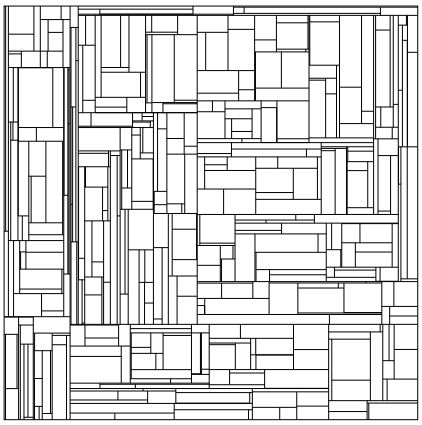
6.2.1 Analysis

The techniques that we have used to compute most of the costs of the previous variants consist in writing a recurrence on one variable assuming that the searching area is always a square. Note that if the area were not a square, until now we could scale it without loss of generality. By contrast, in the squarish variants, the shape of the searching area dictates the discriminant to use, so we cannot scale the current region. As a consequence, the recurrences have an additional parameter (the ratio of the region), which make them much harder to solve.

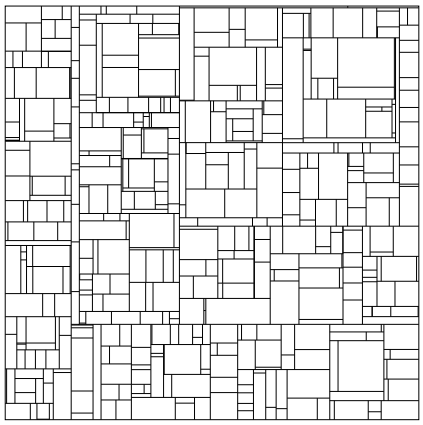
Anyway, we can intuitively (and rigorously) deduce the expected cost of a random search. In a hybrid squarish kd-tree, the discriminant is chosen in a completely independent way of the inserted point, as it is the case of the standard, basic squarish and relaxed variants. Therefore, all these variants have the same the expected cost, that is, $S_n \approx 1.38628 \log_2 n$.

As already seen in the previous chapter, the expected cost of a partial match is connected to the squarish shape of the partitions defined by the kd-tree. Figure 6.4 shows the partitions of the plane for a standard, a squarish and a hybrid squarish kd-tree after inserting the same 500 elements. We can see that the hybrid relaxed kd-tree produces regions less squared than those of the basic squarish variant, and that the shape of the partitions are

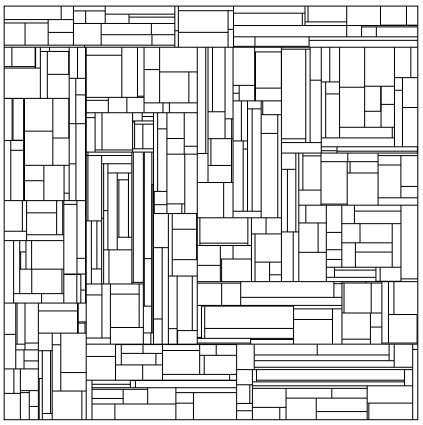
very similar to (and perhaps a bit better than) those of standard kd-trees. Therefore, we can reasonably deduce that the expected cost of a partial match for hybrid squarish kd-trees is similar to that for standard kd-trees. In particular, it must be less efficient for this query than the basic squarish variant.



(a) standard kd-tree



(b) squarish kd-tree



(c) hybrid squarish kd-tree

Figure 6.4: Splits in the plane for a standard, a squarish and a hybrid squarish kd-tree.

We have not proved our conjecture about the expected cost of a partial match operation in a hybrid squarish kd-tree, but the results of the experiments (see Sections 7.1 and 7.2) match well with our hypotheses.

6.3 Hybrid relaxed kd-tree

Finally, the last kd-tree variant that we present in this work is the hybrid relaxed kd-tree. This kd-tree variant chooses the discriminant at random in the first step (as a relaxed kd-tree does), then it chooses another discriminant at random from the $k-1$ discriminants that have not been used yet for the next level, and so on. After k levels, the process starts again. For $k = 2$ this means that every two levels, either x or y is chosen at random, and then the other discriminant is assigned in the following level.

6.3.1 Analysis

Search

The recurrence that describes the expected cost of a completely specified random search in a hybrid relaxed kd-tree is the same that the recurrence for a standard kd-tree. That is,

$$\begin{aligned} S_n &\simeq 1 + \int_0^1 (x \cdot S_{xn} + (1-x) \cdot S_{n-xn})\, dx \\ &\sim 2 \ln n \simeq 1.38629 \log_2 n \end{aligned} \tag{6.3}$$

for any $k \geq 2$.

Taking into account that the expected cost of a search in a standard and a relaxed kd-tree is also $1.38629... \log_2 n$ for $k \geq 2$, it is coherent that this hybrid variant has the same expected cost.

Partial Match

Consider a hybrid relaxed kd-tree in a 2-dimensional space, and suppose that the coordinate x is the one specified in the query. With probability $1/2$, the kd-tree uses x as a discriminant at the first level, and it visits only the appropriate subtree. The probability to proceed the search into the left subtree is x, and the probability to proceed into the right subtree is $1 - x$. With probability $1/2$, the kd-tree uses y as a discriminant, and it proceeds the search visiting both subtrees.

In the next level, the recurrence to use is the standard, either the one for the x discriminant or the one for the y discriminant. Then, the global recurrence that expresses the expected cost of a partial match operation for

a hybrid relaxed kd-tree is

$$\begin{aligned} P_n &\simeq 1 + \frac{1}{2}\int_0^1 \left(x \cdot Y_{xn} + (1-x)\cdot Y_{(1-x)n}\right) dx \\ &+ \frac{1}{2}\int_0^1 \left(X_{yn} + X_{(1-y)n}\right) dy, \end{aligned}$$

where X_n and Y_n are the recurrences for a standard partial match that apply in the second step. These recurrences are of the form

$$\begin{aligned} X_n &\simeq 1 + 2\int_0^1 xP_{xn}\, dx, \\ Y_n &\simeq 1 + 2\int_0^1 P_{yn}\, dy. \end{aligned}$$

Under the hypothesis that $P_n \sim c\, n^\alpha$, we get the equation

$$(\alpha+1)(\alpha+3) = 4,$$

whose solution is

$$\alpha = \frac{\sqrt{17}-3}{2} \simeq 0.56155$$

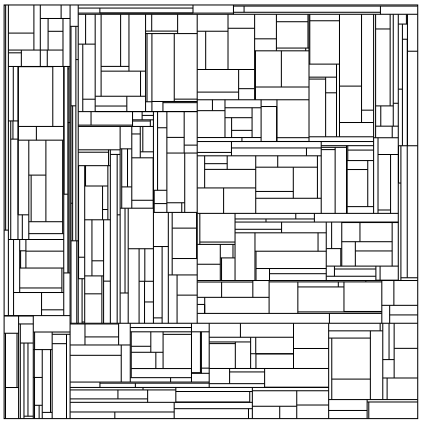
Therefore,

$$P_n = \Theta(n^{0.56155\ldots}). \tag{6.4}$$

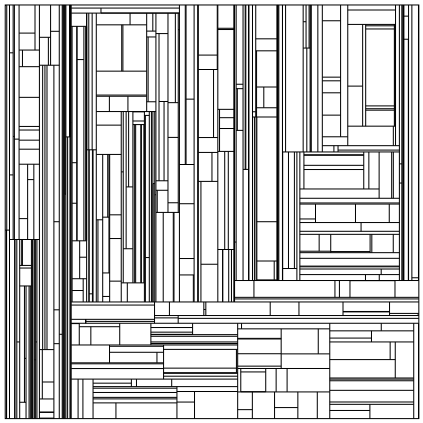
Note that, as could be intuitively expected, the average cost of a partial match for a hybrid relaxed kd-tree coincides with the cost for standard kd-trees (see 3.3).

Figure 6.5 shows the partitions of the plane for a squarish and a hybrid squarish kd-tree after inserting the same 500 elements.

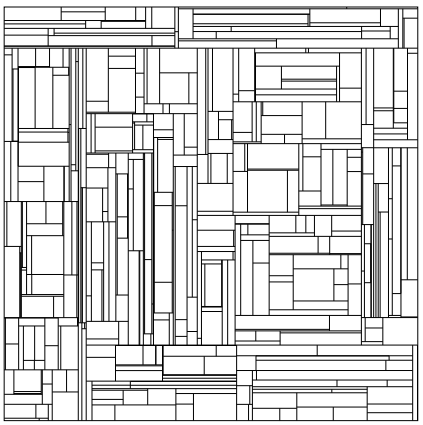
In a hybrid relaxed kd-tree, the regions are less elongated that in a basic relaxed kd-tree, so that its expected cost is lower. Comparing the hybrid relaxed kd-tree and the standard, we can see that the regions in the searching area are very similar (if not mainly equal), which is consistent with the fact that both variants have the same expected cost for a partial match operation.



(a) standard kd-tree



(b) relaxed kd-tree



(c) hybrid relaxed kd-tree

Figure 6.5: Partitions of the plane for a standard, relaxed and hybrid relaxed kd-tree.

Chapter 7

Experimental Work

In this chapter we show and discuss the results of the experiments carried out to test the behaviour and the performance of all the kd-tree variants discussed in this work. These experiments allow us to compare the new proposals with existing variants and confirm the theoretical results obtained in previous chapters.

We have run many experiments on all implemented operations and kd-tree variants, testing several dimensions and sizes. Some of these variants are well known: *standard*, *squarish* and *relaxed* kd-trees. We have tested the new variants proposed in this thesis too. We talk about *median*, *hybrid squarish*, *hybrid median* and *hybrid relaxed* kd-trees. We organize the results in sections corresponding each one to a specific operation.

The experiments have been conducted on random kd-trees, that is, kd-trees generated after inserting n points independently and uniformly distributed on a k-dimensional square, for $k \geq 2$. The random number generator used is based on the method developed by Donald Knuth for the *Stanford GraphBase*. We generate two sets of random numbers to run each experiment. The first one is used to build the kd-tree structure and the second one to run the corresponding operation. The experiment is repeated several times, generating for each one new random inputs. The experiments count the number of visited elements, and compute the averages. These results are plotted in two figures: one with the basic variants (standard, squarish, median and relaxed) and another one with hybrid variants. To compare basic and hybrid variants, we include in the second plot two basic variants as a reference: the standard kd-tree and the more efficient basic variant for the current operation.

In order to facilitate the reading of the results, we always follow the same convention by assigning different colors and line styles for each kd-tree

variant. The conventions that we will use from now on are the following:

- The **black solid line** shows the results for standard kd-tree.
- The **blue solid line** shows the results for squarish kd-tree.
- The **red solid line** shows the results for median kd-tree.
- The **green solid line** shows the results for relaxed kd-tree.
- The **blue dashed line** shows the results for hybrid squarish kd-tree.
- The **red dashed line** shows the results for hybrid median kd-tree.
- The **green dashed line** shows the results for hybrid relaxed kd-tree.

7.1 Search

The cost of a search in all variants of kd-trees is of the form $c \log_2 n$, where n is the size of the tree and c is a constant that depends on the variant of kd-tree, and in some cases on the dimension k. We estimate this constant c experimentally, to make comparisons among the different kd-tree variants.

Figure 7.1 shows the results for searches in 2d-trees of sizes from 1000 to 1000000. For each kd-tree, we have run 10000 searches, and the whole experiment has been repeated 50 times.

The experiment are consistent with the fact that the cost S_n tends to $1.38635... \log_2 n$ for large n for standard, squarish and relaxed kd-trees (see Eqs. 3.8, 4.1 and 4.3 on pages 25–34). In the case of median kd-trees, c has a lower value, around 1.15086, as we have proved in Eq. 5.2 on page 38. Regarding the other variants, hybrid standard and hybrid relaxed kd-trees have $c \approx 1.38635$ too, while hybrid median kd-trees have a constant value around 1.25766 (see Eq. 6.1 on page 47), halfway between the constants corresponding to standard and to median kd-trees.

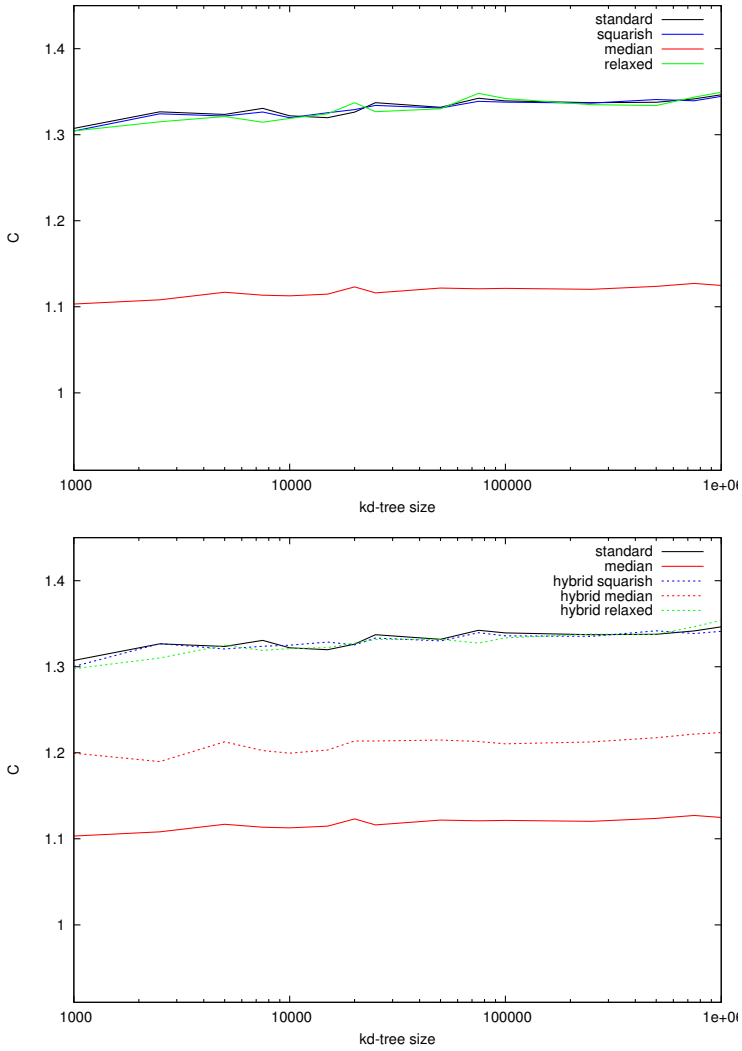


Figure 7.1: Search: value of c for 2-dimensional kd-tree.

In the second experiment (Figure 7.2), we have run searches in kd-trees with different dimensions so to study the dependence of c with respect to the number of dimensions k. We have run 10000 searches in kd-trees of size 100000 elements, and we have tested from 2 to 1000 dimensions. For each dimension we have run 100 experiments.

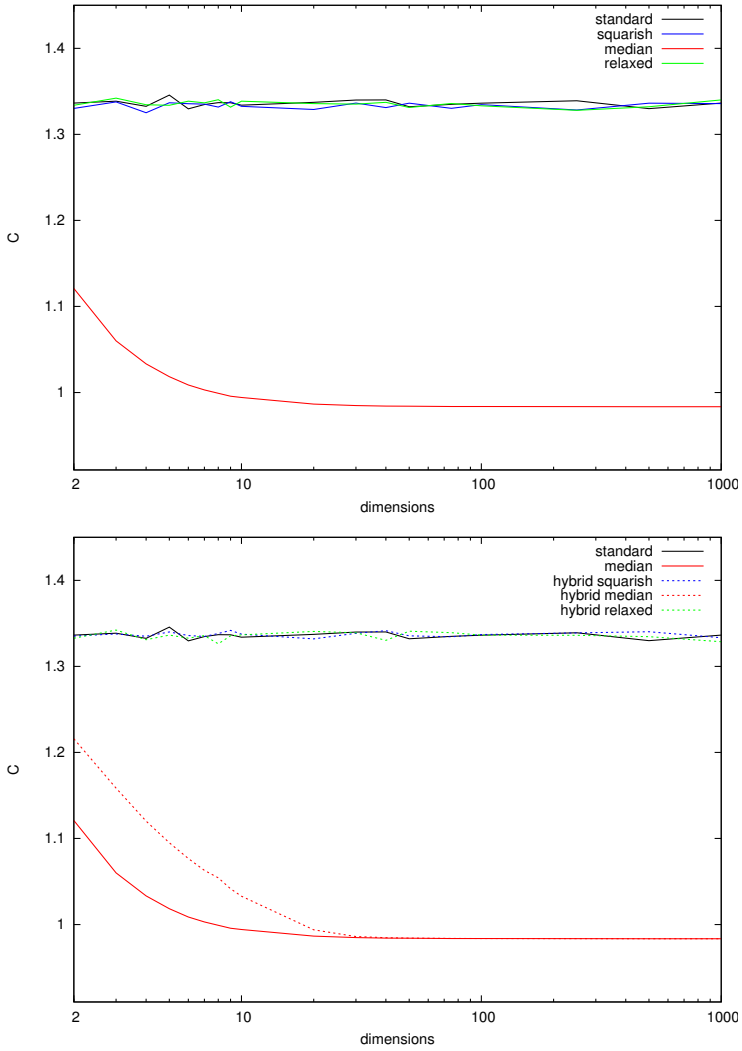


Figure 7.2: Search: value of c for several dimensions.

We can see that, except for the median and hybrid median variants, the constant c is independent from the number of dimensions of the kd-tree, and $c \approx 1.38635$ whatever dimensions we have. However, the median variants have better performance for larger dimensions, and the constant c tends to the optimal 1 when we increase k, coherently with Eq. 5.3 on page 40.

7.2 Partial Match

The expected cost of a partial match in a kd-tree of size n is $P_n = \beta\, n^\alpha$, where α is a value between 0 and 1 that depends on the dimension and on the variant of kd-tree that we consider. To experimentally estimate the value of α we need to work with large values of n and to get rid of the constant β. In particular, we use the quotients of P_n with respect to P_{1000}, so that

$$\begin{aligned} \frac{P_{1000}}{P_n} &\simeq \frac{\beta \cdot 1000^\alpha}{\beta \cdot n^\alpha}, \\ \alpha &\simeq \frac{\ln \frac{P_{1000}}{P_n}}{\ln\left(\frac{1000}{n}\right)}. \end{aligned}$$

In the first experiment, whose results are presented in Figure 7.3, we have run partial matches in 2d-trees of sizes ranging from 1000 to 500000. For each kd-tree, we have run 1000 partial match searches specifying x and 1000 partial match searches specifying y. We have done this to get the average of a partial match independently of the specified coordinate. The experiment was repeated 100 times.

The experimental results match well the theoretical results:

- standard kd-tree: $P_n = \Theta(n^{0.56155\ldots})$ (Eq. 3.3, page 24)
- squarish kd-tree: $P_n = \Theta(n^{0.5})$ (Eq. 4.2, page 32)
- median kd-tree: $P_n = \Theta(n^{0.60196\ldots})$ (Eq. 5.4, page 42)
- relaxed kd-tree: $P_n = \Theta(n^{0.61803\ldots})$ (Eq. 4.4, page 34)
- hybrid median kd-tree: $P_n = \Theta(n^{0.54595\ldots})$ (Eq. 6.2, page 49)
- hybrid relaxed kd-tree: $P_n = \Theta(n^{0.61803\ldots})$ (Eq. 6.4, page 54)

About the hybrid squarish kd-tree, we have not computed the theoretical value for α, but the experiments seem to show that it is only a bit better than a standard kd-tree, and hence it is worse than the basic squarish kd-tree.

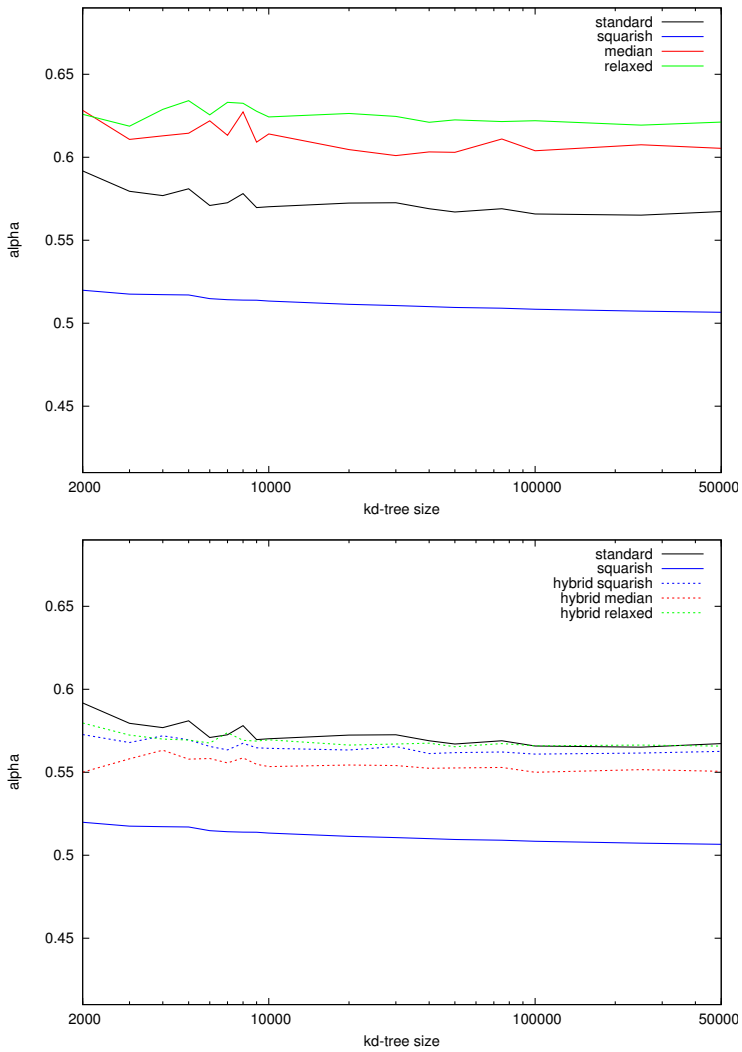


Figure 7.3: Partial Match: value of α for 2-dimensional kd-trees.

In the second experiment, we check the dependence of the expected cost of a partial match in a 2d-tree with respect to the specified dimension. We have built kd-trees of sizes from 1000 to 500000, and for each kd-tree we have run 10000 partial matches specifying x and another separate set of 10000 partial matches specifying y. We have repeated the experiment 200 times for each size.

The results are shown in Figures 7.4, 7.5 and 7.6 for the standard, squarish and median kd-trees respectively. We observe that, as expected, except for the standard kd-tree, it does not matter which dimension is specified for the partial match search. In other words, the constant β in the cost of

a partial match depends on the query pattern,$(x, *)$ or $(*, y)$, for standard kd-trees, but is independent of the pattern in the other variants. We have not included the graphics for the relaxed and the hybrid versions because the results are the same and the number of elements visited does not depend on the specified dimension either.

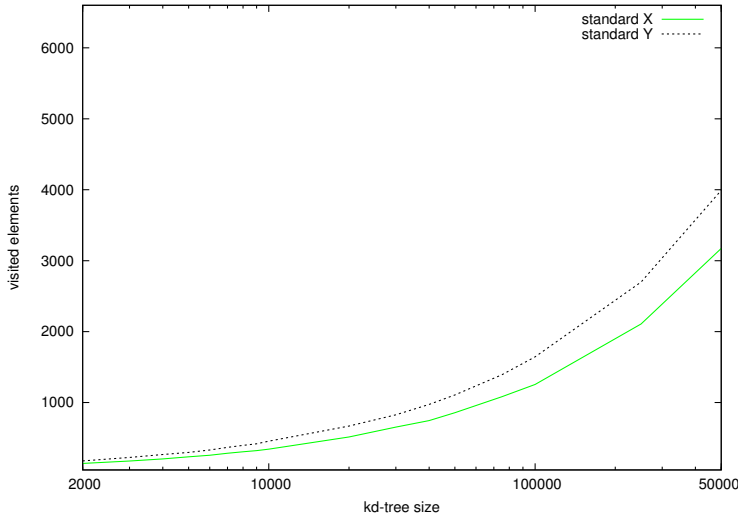


Figure 7.4: Partial Match: number of elements visited depending on the fixed dimension in a standard kd-tree.

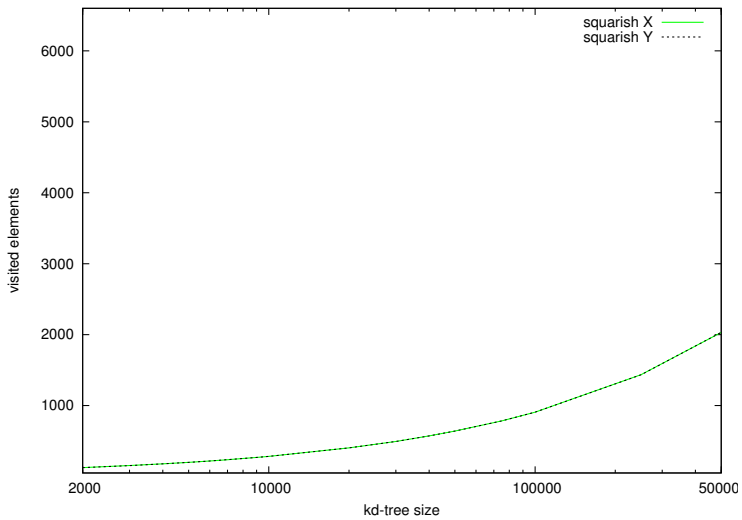


Figure 7.5: Partial Match: number of elements visited depending on the fixed dimension in a squarish kd-tree.

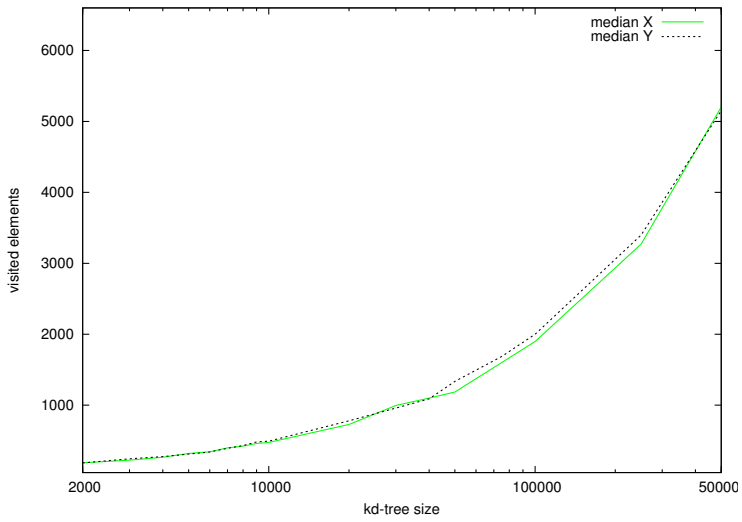


Figure 7.6: Partial Match: number of elements visited depending on the fixed dimension in a median kd-tree.

Regarding the standard kd-tree, the experiment shows that the constant c is around 28% larger when the dimension fixed is y. This matches well with the theoretical result (see Eq. 3.4 on page 24). The reason for this different behaviour is that the standard variant always uses x as the discriminant at the root.

Let us analyze the dependence of α with respect to the number of specified and unspecified dimensions in a partial match. We have run 500 partial match searches in kd-trees of size 10000, and we have tested from 2 to 100 dimensions, running for each dimension 50 experiments.

In Figure 7.7 we have the results of the experiments with only one specified dimension and $k-1$ unspecified dimensions. We can see that if the number of dimensions grows, α is near 1, so that $n^\alpha \approx n$. This makes sense because the algorithm only discards a subtree when the discriminant at the current node is specified, which happens, on the average, 1 over every k times. Consequently, the algorithm visits almost all the points.

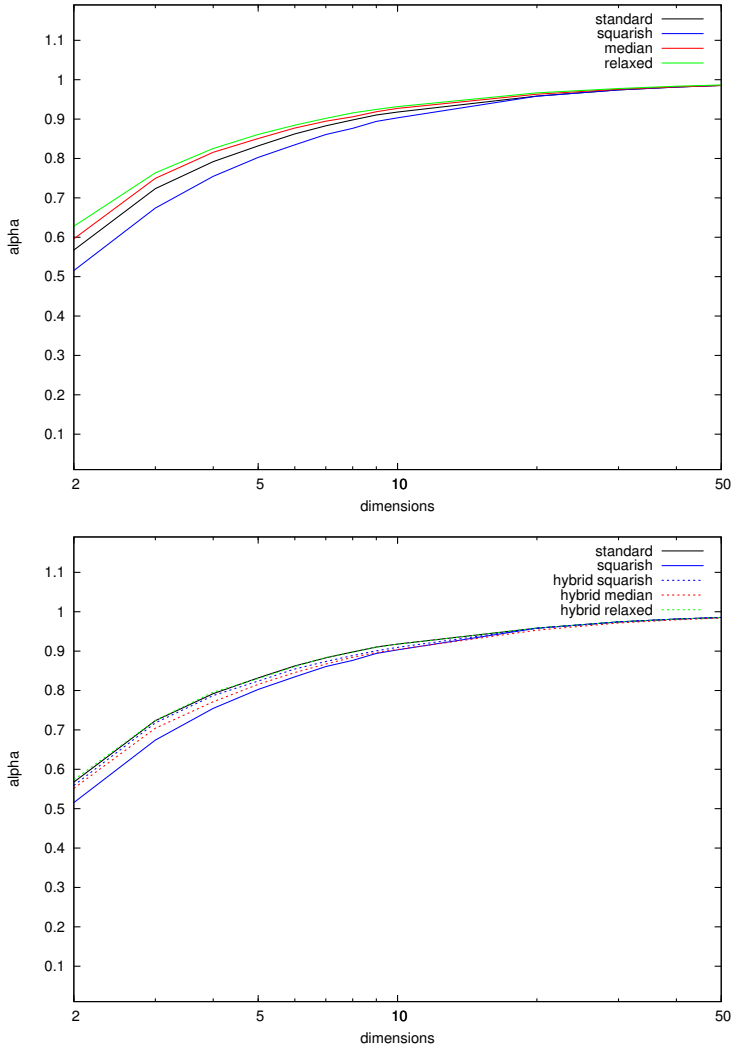


Figure 7.7: Partial match for growing dimensions: value of α with only one specified dimension.

Figure 7.8 is the opposite situation, where there is only one unspecified dimension. For all the variants, the more dimensions are specified, the closer is α to zero. If we fix all the dimensions, we are running an exact search and this operation has cost $\Theta(\log n)$.

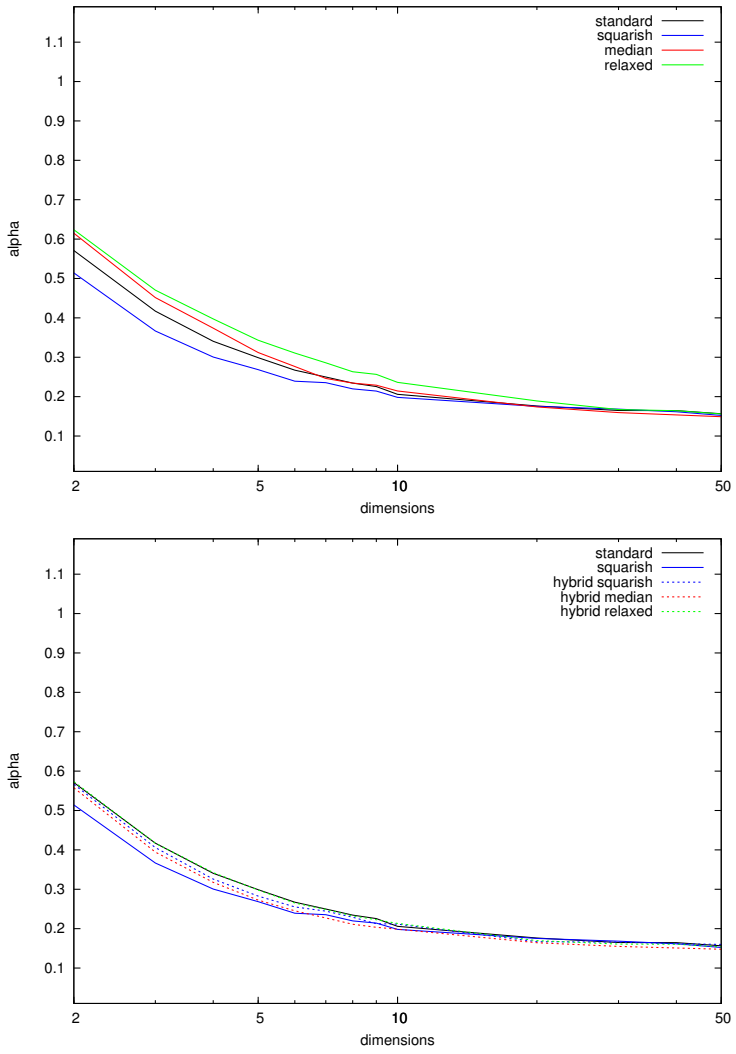


Figure 7.8: Partial match for growing dimensions: value of α with all dimensions specified except one.

We must take into account that all the theoretical results are in the limit, when we would have a kd-tree with a large number of elements and dimensions. Because of the fact that we deal with finite values for these parameters, we only see the tendency for α. But this seems enough to confirm our deductions.

7.3 Linear Match

For the linear match experiments, we have considered two models. In the first one, we fix a slope and we choose a point of the region at random. Then, the probability of choosing a certain line is proportional to the length of the line that intersects the bounding box. This model seems very difficult to analyze.

The second model is the one analyzed in Chapter 3 on page 26. Here, we fix a slope and choose any line with that slope that intersects the bounding box with equal probability.

For both models, we have chosen these slopes to check: {0, 0.1, 0.5, 1, 2, 10, 1000}.

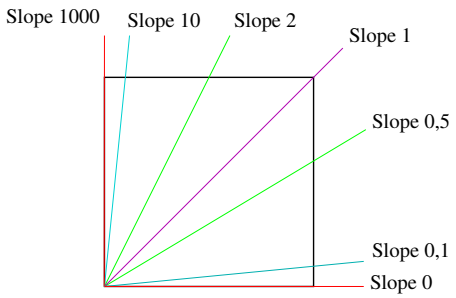


Figure 7.9: Slopes used to run the linear match.

These slopes cover a wide spectrum of possible lines slopes. We have only used positive slopes because, by symmetry, the results can be extrapolated to negative values. Note that a line with slope 0 corresponds to a partial match when the y coordinate is specified; a line with slope $+\infty$ corresponds to a partial match when the x coordinate is specified.

In the experiment for the first model, we have created a 2-dimensional kd-tree with 10000 elements. We have generated 10000 random points and we have fixed the first slope. With each one of these points and the slope we have run a linear match. Afterwards, we have fixed the second slope, and using the same points, we have run 10000 more queries, and so on, until we have checked all the slopes. This process have been repeated 2300 times.

Figure 7.10 shows the results of the linear match query experiment. In the plot, it has been used the corresponding sexagesimal degrees instead of the slope value because this way the graphic is clearer.

We can see that, except for standard kd-trees, the plot is symmetric respect the slope, that is, the number of elements visited is the same when we fix slope 0 and slope $+\infty$. These slopes corresponds to a partial match,

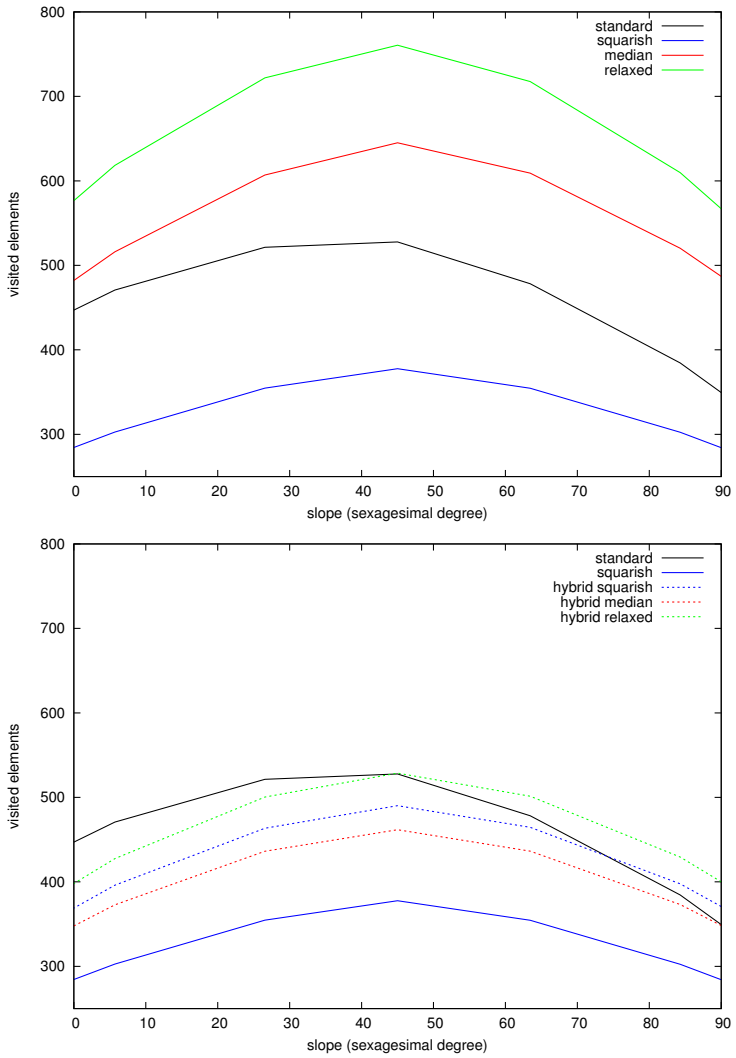


Figure 7.10: Linear Match: visited elements depending on the line slope.

which has the same cost independently of the dimension specified (in the previous section it was explained the reason for that). As it is expected, the standard kd-tree visits more elements with slope $+\infty$, that is, a partial match with y specified is worse than a partial match with x specified.

We can observe as well that the maximum number of visited elements occurs when the slope is 1 (or 45 degrees). Remember that the probability of choosing a line depends on the length of the line that intersects with the region. Hence, the lines passing near the central region, which are the longest possible lines (and therefore are the lines that intersect with more bounding boxes and visit more points), are also more likely to be the query

lines. This could be a possible explanation of this phenomenon. Another complementary reason could be that a 45-degree slope is the one that worse fits the 0-degrees and 90-degrees cuts of the space made by the 2d-trees.

Comparing the different variants, the squarish kd-tree is the more efficient, followed by standard, median and relaxed kd-trees, in this order. As for the hybrid variants, the most efficient if the hybrid median kd-tree, which improves the basic median, and its performance is between that of the squarish and that of standard kd-trees. As expected, these results match the ones we got for the partial match query.

In the experiments for the second model, we work with the scenario analyzed in Section 3. We have build a 2d-tree of size 10000, we have fixed the slope and then we have run 1000 linear matches, choosing for each one a random line over all the possible lines that cross the space. The experiment was repeated 1000 times for each slope, and covered slopes from 0 to 1000.

The results of the experiment are consistent with our theoretical analysis. The linear match with slope 0 and slope $+\infty$ correspond to a partial match specifying the y and the x coordinate, respectively, and also match the partial match results. The squarish kd-tree is the most efficient variant for this query, followed by the hybrid median kd-tree, and the relaxed kd-tree is the less efficient.

The behaviour for the standard kd-tree agrees with the plot in Figure 3.1 (page 28). Moreover, it is consistent with the values that we got in the analysis for the partial match 3.4 (page 24), where it was shown that the number of elements visited by a partial match that specifies the y coordinate (a linear match with slope 0), is around a 28% larger. In the other variants, the number of elements visited seems to be independent of the slope of the line.

7.4 Orthogonal Range

The expected cost of an orthogonal range depends on the number of points reported, the expected cost of a partial match and the expected cost of a search (see Eq. 3.13 on page 29). We can state that, for large values of n, the dominant term of the expression is $\Theta(n^{\alpha})$ if the range search is "small enough". In other words, the behaviour of the orthogonal range has some relation with the behaviour of a partial match.

Since the formula for the cost is very complex, we cannot isolate the parameter α from our experimental results. Therefore, the plots presented

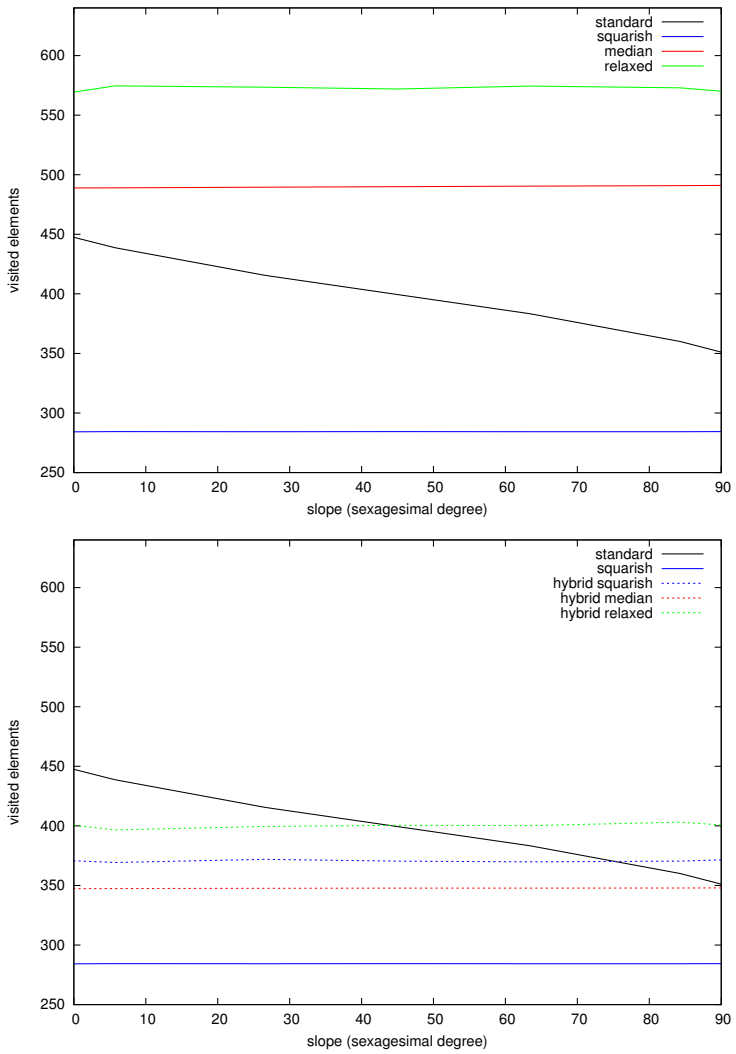


Figure 7.11: Linear Match: visited elements depending on the line slope.

in this section only show the number of visited elements with respect to the size of the kd-tree.

Figures 7.12 and 7.13 include the results for orthogonal range searches for 2d-trees. In our experiments, the input rectangle is always a square. These two experiments differ only in the size of the squares: 0.001 in the first one and 0.005 in the second one. We have tested 2d-trees of sizes from 1000 to 500000. For each size, we have built 100 different kd-trees, and over each one of these kd-trees we have run 10000 orthogonal range searches.

We observe that, for small n, the median variants seem to be the most efficient. The reason could be that these kd-trees have the lowest cost for the

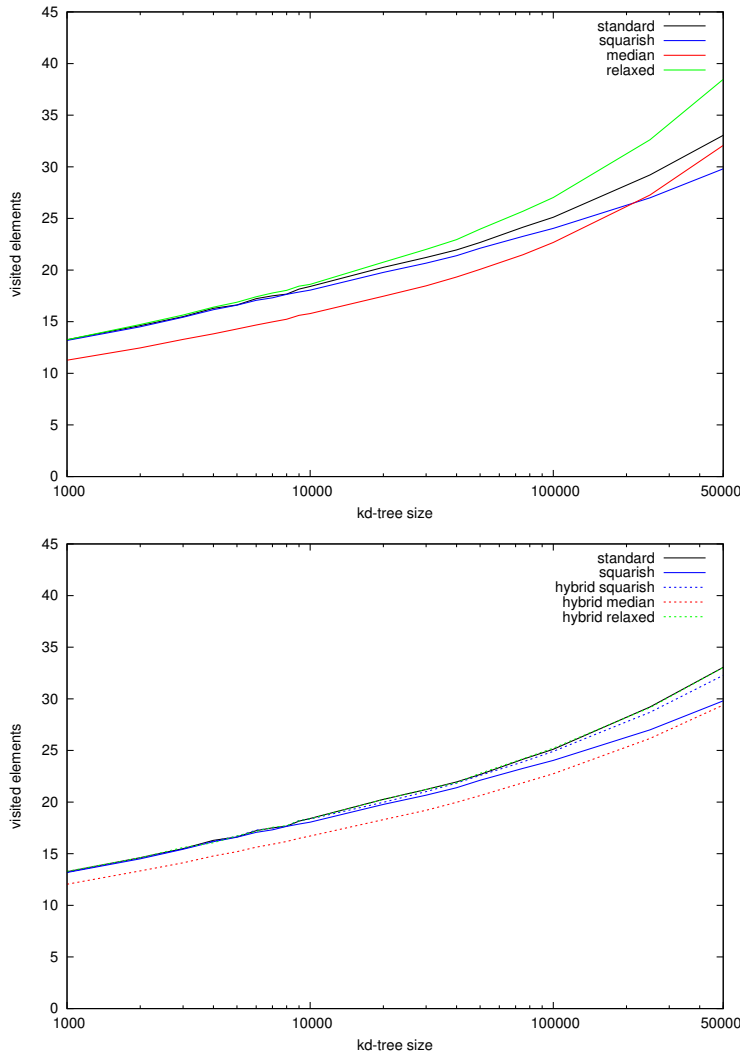


Figure 7.12: Orthogonal Range with query edge length 0.001: number of visited elements.

search operation, and we have previously seen that the cost of the orthogonal range has a term related to the search cost. Anyway, the dominant term has the partial match cost order, so that in the limit, the behaviour of a orthogonal range query in all the kd-tree variants is similar to that of a partial match.

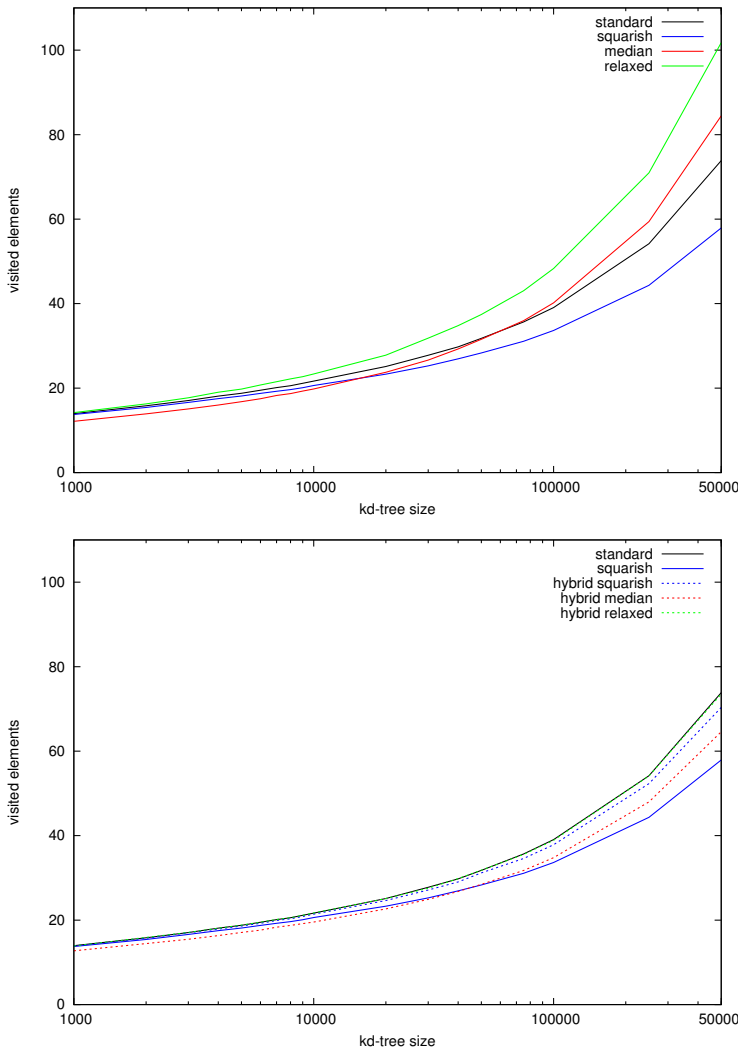


Figure 7.13: Orthogonal Range with query edge length 0.005: number of visited elements.

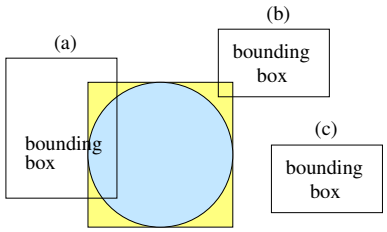
The squarish kd-tree is the most efficient variant, followed by the hybrid median. The other hybrid variants have a performance similar to the standard kd-tree, and the median and the relaxed variants have the worst performance.

7.5 Radius Range

The algorithm for the radius range search, explained in Section 2.6, returns all the points that satisfy the distance constraint. It requires to analyze a point when its bounding box intersects with the region defined by this distance constraint (a ball with the Euclidean distance or a rhombus with the Manhattan distance).

Since the region with the satisfactory points can always be inscribed in a square, and the kd-tree structure allows easily to know if two rectangles intersect, our implementation includes an optimization. The algorithm only analyzes that subtrees whose bounding box intersects with the square where it is inscribed the input region. Although, the algorithm visits more subtrees, it decides which subtrees to analyze in a very efficient way.

In the next figure, the (a) and (c) subtrees are correctly analyzed and discarded respectively. But the (b) subtree is analyzed although its bounding box does not intersect with the satisfactory region. Note that this is the unique situation where a subtree is analyzed without being necessary.



We have not included an experiment fixing the radius and increasing the size of the kd-trees, because the number of elements that would visit the radius range is exactly the same than the elements visited by the orthogonal range, that has been analyzed in the previous section.

Then, in the first scenario (Figure 7.14), we fix the number of elements and increase the size of the radius. We create a kd-tree with 10000 elements and, after defining a radius, we run 10000 different radius range operations over the same kd-tree. The defined radius goes from 0.001 to 0.025, and the distance function used is the Euclidean distance. The experiment is repeated 100 times using the optimized algorithm.

Since the expected cost of the radius range operation is similar to the cost of the orthogonal range, we can deduce that it depends on the number of points reported. Then, as we increase the size of the radius, the number of points to be returned increases too. We observe this behaviour in our experiment, where the expected cost of the operation is a lineal function.

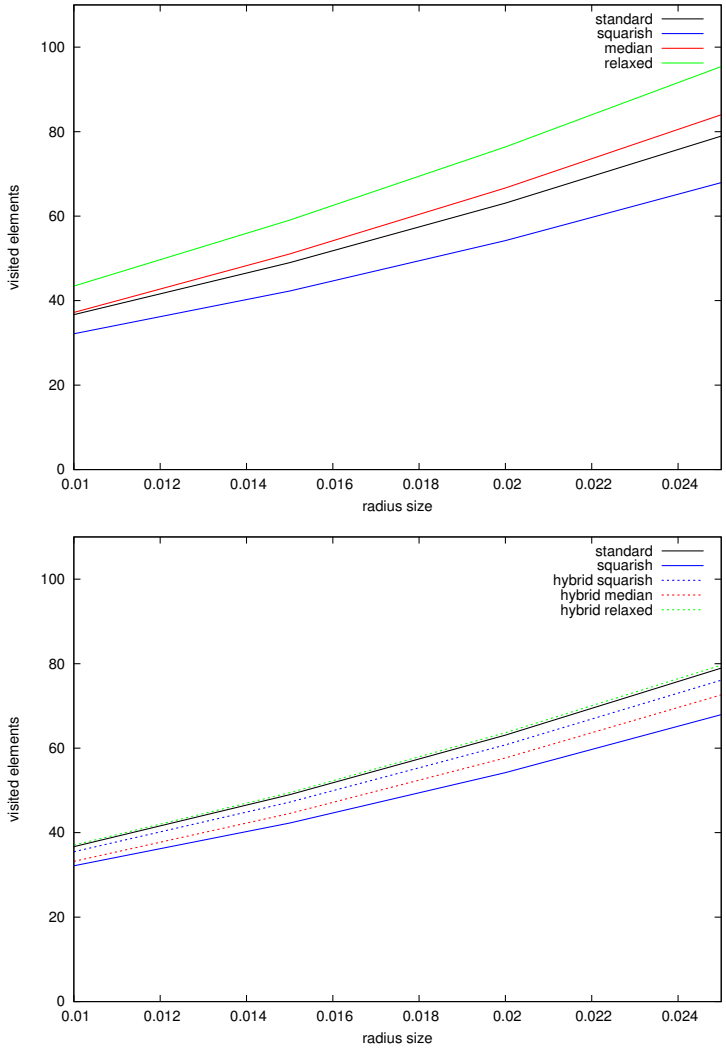
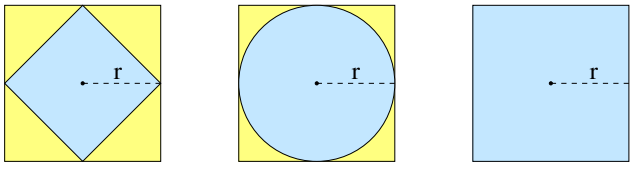


Figure 7.14: Radius Range: number of visited elements.

In the next experiment, we have tested three distance functions: Manhattan, Euclidean and Chebyshev (or infinite distance). The following figure shows in blue the location of the points that satisfy the distance constraint (the distance between them and a center point is less than a radius r).



Manhattan Euclidean Chebyshev

The optimized radius range algorithm analyzes the same number of points than an orthogonal range, because it discards the elements using the square where the circle is inscribed. The points whose region intersects with the yellow area but not intersects with the blue area are not discarded. Then, their subtrees are recursively analyzed, although they can not contain points that satisfy the distance constraint.

Therefore, the basic radius range algorithm only analyzes the points whose region overlaps with the area that satisfies the distance constraints. That is, all points whose region only intersects with the yellow area are discarded.

The following plots (Figures 7.15, 7.16, 7.17) show the results of an experiment that compares the basic and the optimized radius range algorithm. As in the previous experiment, the radius size goes from 0.001 to 0.025. We have tested 100 different 2d-trees of size 10000 and we have run 10000 orthogonal range to each one. The distance function used are Manhattan, Euclidean and Chebyshev.

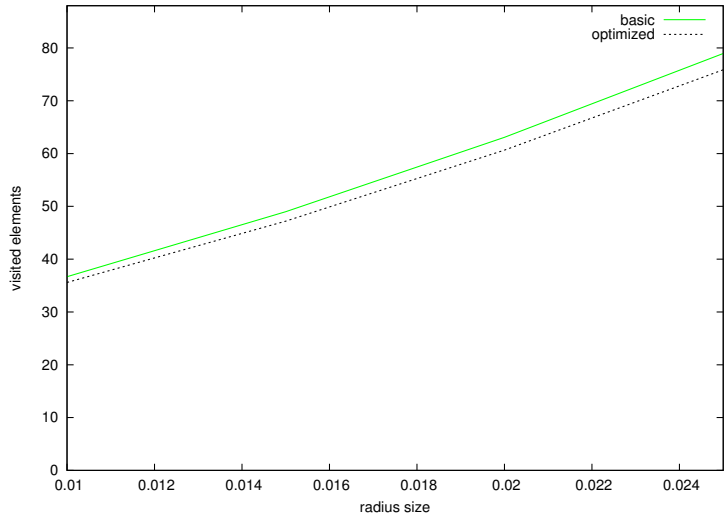


Figure 7.15: Radius Range Manhattan distance: number of elements visited in a median kd-tree.

We have only included the plots related to the median variant, because the other variants follow the same pattern. As it is expected, we can see that the optimized version visits less elements. Logically, this improvement is related to the size of the yellow area. Using the Manhattan distance we get some improvement which is reduced when we use the Euclidean distance. Regarding to the Chebyshev distance, the number of elements visited is the same in both versions, because they are equivalent.

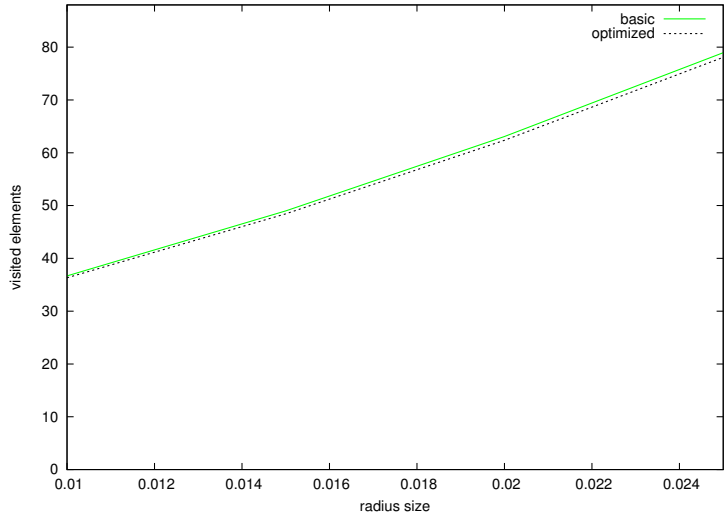


Figure 7.16: Radius Range Euclidean distance: number of elements visited in a median kd-tree.

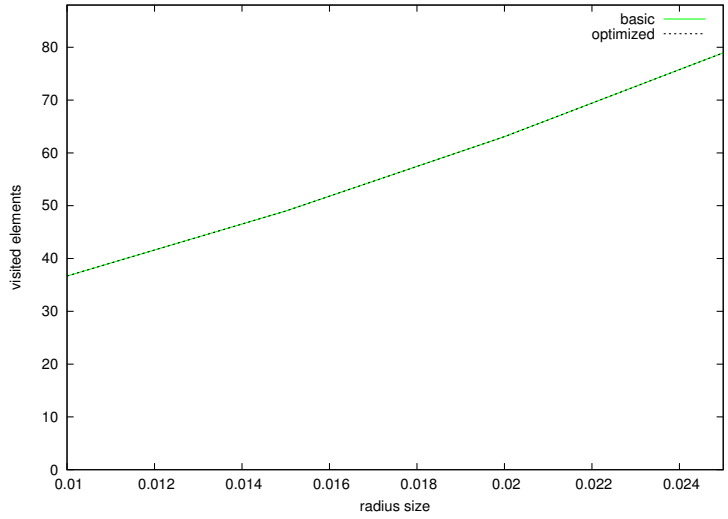


Figure 7.17: Radius Range Chebyshev distance: number of elements visited in a median kd-tree.

Anyway, in order to discard a point within the yellow area, the algorithm has to analyze its bounding box. Hence, it has to run some computations, and this improvement on the number of visited elements is not necessarily reflected in the running time.

7.6 Nearest Neighbor

The nearest neighbor algorithm has the same expected cost as the radius range and the orthogonal range, and the results of the experiments shown in Figures 7.18 and 7.19 confirm it.

The experiment builds kd-trees of sizes from 1000 to 500000 and it runs 1000 nearest neighbor searches. This experiment is repeated 50 times.

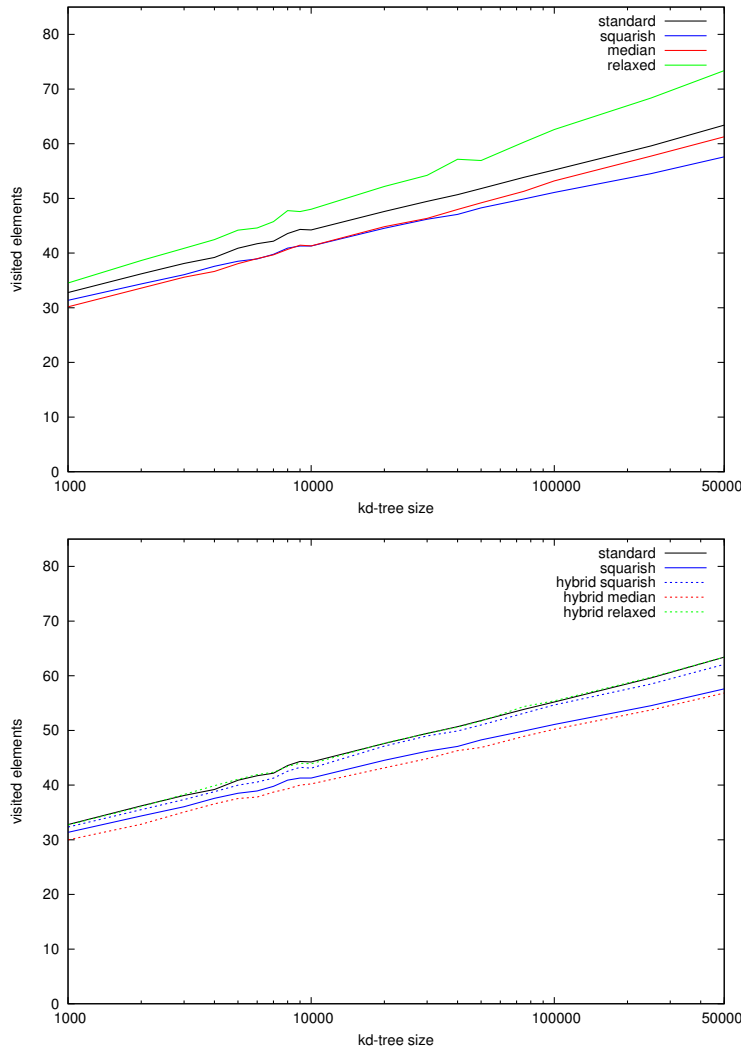


Figure 7.18: Nearest Neighbor: just one neighbor.

Figure 7.18 shows the average number of visited elements when the algorithm searches the closest element. On the other hand, Figure 7.19 searches the closest element, the second closest and so on until, it gets the 100 near-

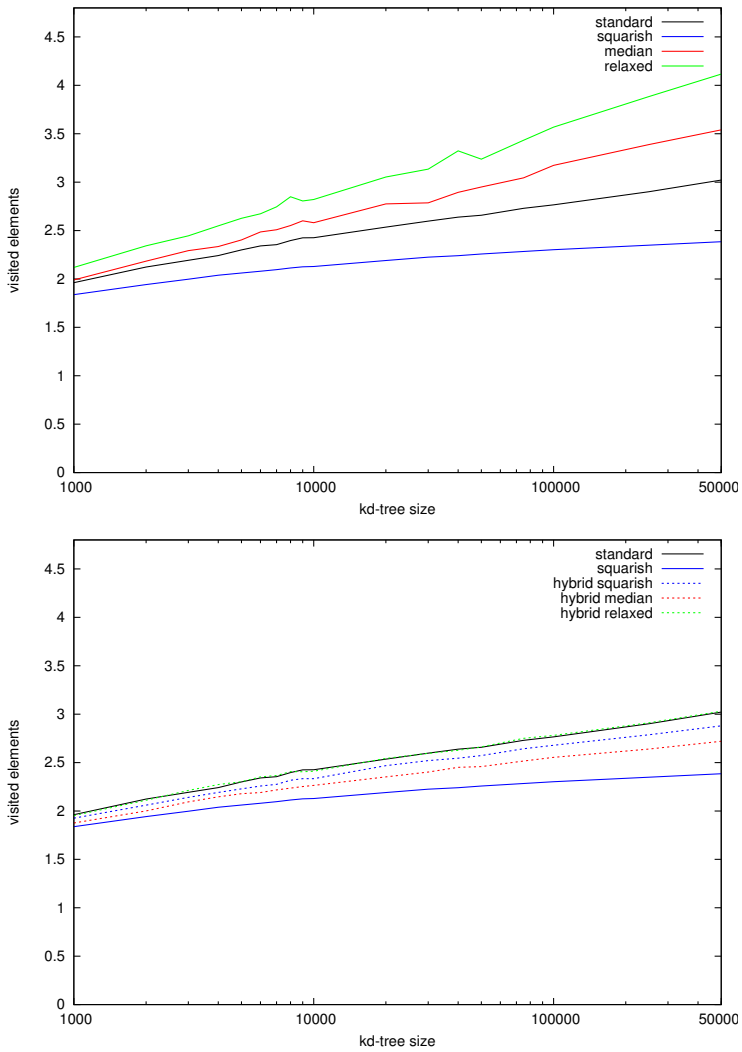


Figure 7.19: Nearest Neighbor: 100 closest neighbors.

est neighbor elements. We can see that the number of visited elements is much lower in the second case because the algorithm has to analyze many elements when it searches the closest point, but it takes advantage of these visits in the subsequent searches.

Chapter 8

Spatial and Metric Template Library

One of the goals of this thesis is to develop and implement an efficient library of generic metric and spatial data structures and algorithms in the spirit of the C++ *Standard Template Library*, STL.

Studying other existing spatial and metric libraries, we have found two libraries that are similar to ours. The first one, a library developed by Figueroa, Navarro and Chávez [FNC08] comes from the conference *Similarity Search and Applications* (SISAP). It is an open-source C-based library, but this library is not implemented in C++ and it does not follow the philosophy of the STL.

On the other hand, we have the ANN library, implemented by D. Mount and S. Arya [Mou10]. ANN stands for *Approximate Nearest Neighbor* and it is a library written in C++ but designed to work with static data, that is, the information that has to be stored in the data structure is always known in advance. Moreover, this library does not follow the principles of the STL and it does not use iterators to return results.

Our library, that we name *Spatial and Metric Template Library*, SMTL for short, is a library developed in C++, following principles and architecture of the STL. The fundamental components of SMTL are the *container* classes, whose purpose is to store multidimensional data and to provide several operations to retrieve information, and the *iterators*, to manipulate the results of the queries. Both components are designed to work well together with STL components, as they are based in the same set of principles as the STL library.

The components of the SMTL library are robust, flexible and have been throughly tested for performance tuning, providing ready-to-use solutions to

common problems with multidimensional data, like nearest neighbor search or orthogonal range search.

In the following sections we will explain the use of the library through some examples and give a full list of all the components in the library (Section 8.1) and later we discuss briefly some aspects of the implementation of the library (Section 8.2).

8.1 Using the SMTL

The SMTL library is available in `http://www.lsi.upc.edu/~mpons`

8.1.1 Containers

The SMTL offers the seven kd-tree variants described in previous chapters:

- `std_kdtree` (standard)
- `sqr_kdtree` (squarish)
- `rlx_kdtree` (relaxed)
- `mdn_kdtree` (median)
- `hybsqr_kdtree` (hybrid squarish)
- `hybrlx_kdtree` (hybrid relaxed)
- `hybmdn_kdtree` (hybrid median)

All those classes are parameterized by the type of the keys, the type of the values associated to the keys and the dimension of the keys. This last parameter might be omitted, then the dimension is deduced from the keys. It is assumed that the type of the keys offers an operation `size()` returning the number of attributes (coordinates, dimensions) of the keys and indexing operator `operator[](int i)` returning the i-th attribute of a given key. Thus, if `x` is a key then `x.size()` returns the dimensionality of the key and `x[i]` returns the i-th coordinate of `x`.

All the containers have exactly the same functionality and have equivalent methods, so we describe now only `std_kdtree`. Furthermore, all the seven container classes derive from the abstract class `kdtree` that supports all the consulting methods.

```
template < typename Key,
           typename Value,
           unsigned int Dim=0 >
class std_kdtree : public kdtree<Key,Value,Dim>
{
public:
  std_kdtree( unsigned int dim,
              const Key& minv=Key(),
              const Key& maxv=Key() );

  std_kdtree( const Key& minv=Key(),
              const Key& maxv=Key() );

  template <typename InputIterator>
  std_kdtree( unsigned int dim,
              InputIterator beg,
              InputIterator end,
              const Key& minv=Key(),
              const Key& maxv=Key() );

  template <typename InputIterator>
  std_kdtree( InputIterator beg,
              InputIterator end,
              const Key& minv=Key(),
              const Key& maxv=Key() );

  void insert( const Key& k,
               const Value& v );
}
```

All the kd-trees are generic. The type of the keys and values are parameters of the template as well as the dimension of the keys. If the dimension parameter of the template is not defined, then the parameter `dim` of the constructor is used. And if both are defined but they are inconsistent, the dimension specified in the constructor prevails. Moreover, when it is not defined any dimension, the dimension of the first element inserted is taken as a dimension for the kd-tree.

Another aspect to mention is that if we insert one element with dimension larger than the kd-tree dimension, they are only used the number of dimensions specified by the kd-tree. But if the element to insert has dimension smaller than the kd-tree dimension an exception is thrown.

The first constructor creates an empty kd-tree where the dimensionality of the keys is defined by the first parameter. The second and the third parameters are optionals and correspond to the elements that define the global bounding box. If these parameters are not defined, the default value of the type `key` is taken. In any case, the global bounding box is resized whenever we insert one point in the kd-tree falling outside.

The second constructor works as the first one, but in this case the dimension of the keys is not given and it is deduced from the first element inserted in the tree.

The other two constructors create a kd-tree with the elements (pairs <Key,Value>) coming from a sequence. The sequence is defined by two iterators `beg` and `end`, and the algorithm goes through this sequence inserting the elements until it reaches the `end` iterator. There are not any preprocess of the information in order to create a more efficient organization of the data. In the third constructor the dimension is given as a parameter by the user, and in the fourth it is deduced after the first insertion, just like in the other two constructors.

Finally, the `insert` operation inserts the element with key `k` and value `v` in the kd-tree, following the insertion rules of the data structure. This implementation does not allow duplicate elements, so that the operation has no effect when the user wants to insert an element whose key is already in the kd-tree.

Although the constructors of the kd-tree and the insert operation are defined in the specific kd-tree, the remove operation is defined in the abstract class `kdtree`. This operation searches the element with key `k` in the kd-tree and, deletes it from the data structure. Then, the subtree whose root is the removed element is rebuilt without this element.

```
template <typename Key,
          typename Value,
          unsigned int Dim=0>
class kdtree{
public:
  void remove( const Key& key );
  ...
}
```

8.1.2 Queries

Since all the consulting algorithms are identical in all these seven kd-tree variants, we have derived them from the abstract class `kdtree`, that implements all the associative queries. The consulting methods of each specific kd-tree are inherited from the abstract class, which we now review.

```
template <typename Key,
          typename Value,
          unsigned int Dim=0>
class kdtree{
public:
  virtual void insert( const Key& k,
                       const Value& v ) = 0;
```

```
  basic_iterator<Key,Value,Dim> begin() const;
  end_iterator<Key,Value,Dim> end() const;

  srch_iterator<Key,Value,Dim>
      search( const Key& k ) const;

  or_iterator<Key,Value,Dim>
      orthogonal_range_query( const Key& lb,
                              const Key& hb ) const;
  or_iterator<Key,Value,Dim>
      orthogonal_range_query( const Key& center,
                              const vector<double>& side
                            ) const;

  pm_iterator<Key,Value,Dim>
      partial_match_query( const Key& k,
                           const vector<bool>& coords
                         ) const;

  lm_iterator<Key,Value,Dim>
      linear_match_query( const Key& key1,
                          const Key& key2 ) const;
  lm_iterator<Key,Value,Dim>
      linear_match_query( const Key& key,
                          const double slope ) const;

  lb_iterator<Key,Value,Dim>
      within_linear_band_query( const Key& key1,
                                const Key& key2,
                                const double width ) const;
  lb_iterator<Key,Value,Dim>
      within_linear_band_query( const Key& key,
                                const double slope,
                                const double width ) const;

  rr_iterator<Key,Value,Dim>
      radius_range_query( const Key& center,
                          const double radius,
          double (*distance)(const Key&, const Key&)=NULL
                        ) const;
  rr_iterator<Key,Value,Dim>
      radius_range_query_opt( const Key& center,
                              const double radius,
          double (*distance)(const Key&, const Key&)=NULL
                        ) const;

  nn_iterator<Key,Value,Dim>
      nearest_neighbor_query( const Key& key,
          double (*distance)(const Key&, const Key&)=NULL
                              ) const;
};
```

The `insert` operation is a virtual function and it is implemented by the derived classes of `kdtree`.

The other operations are queries that answer the associative retrieval problems. They return a sequence of the objects that match the query, via an iterator. When there are not elements satisfying the query, they return a empty iterator, (`end()`) to indicate that the sequence is empty. We detail the usage of the iterator classes in Section 8.1.3. The list of queries supported by the kd-trees is:

`begin` It returns an iterator to the pair `<k,v>` stored at the root of the kd-tree. This operation is the one used when we want to traverse the elements in the kd-tree, because it returns an interator to the first element and then, incrementing the interator, we get the others by levels, like in a BFS algorithm.

`end` It returns an empty iterator.

`search` Given an object `k`, it returns an iterator to the pair `<k,v>` where `v` is the value associated to the key `k` in the data structure, if it is present; otherwise, it returns an empty iterator.

The following operations return an iterator with all the pairs `<k,v>` that match the query. This sequence is not computed in advance, and the following element is always searched when the user asks for it. Each iterator stores some information in order to be more efficient to move to the next element in the sequence.

`orthogonal_range_query` The input is a region of the space that can be defined by the two extremal corners (`lb` and `hb`) or by a center key `center` and a vector of doubles `side` that indicates the lenght of the region for each specific coordinate[1]. Thus, the dimension of the vector `side` is the same than the dimension of the keys. We provide two methods to support these two possibilities. The query returns an iterator to the beginning of the sequence of all pairs `<k,v>` such that `k` lies within the specified region.

`partial_match_query` The input parameters are a key `k` and a vector of booleans `coords` that indicates which coordinates of the key are specified and which are irrelevant. It returns an iterator to the beginning of the sequence of all pairs `<k',v>` such that `k'` matches the query described by `k` and `coords`.

[1] this second version of the orthogonal range query only works for numerical keys

`linear_match_query` The input is a line that can be defined by two keys or by a key and a slope. We provide two methods to support these two possibilities. This query can only be used in a 2-dimensional space for numerical keys. It returns an iterator to the beginning of the sequence of all pairs `<k,v>` such that `k` belongs to the given line.

`within_linear_band_query` The input is a line that can be defined by two keys or by a key and a slope, and a certain `width`. As in the previous query, we provide two methods to support these two possibilities. This query can only be used in a 2-dimensional space for numerical keys. It returns an iterator to the beginning of the sequence of all pairs `<k,v>` such that `k` is, at the most, to a distance `radius` to the given line.

`radius_range_query` The input is a key `center`, a radius `radius` and a certain distance function `distance`. Then, the query returns an iterator to the beginning of the sequence of all pairs `<k,v>` such that `k` is, at the most, to a distance `radius` from `center`, where this distance is computed using the `distance` function. The distance parameter is optional and it is taken the Euclidean distance when it is not specified.

`radius_range_query_opt` This query returns the sequence of all pairs `<k,v>` that lies within a region defined by a `center`, a `radius` and a `distance` function. This is the same results that the `radius_range_query`, but this query applies the optimization described in Section 7.5.

`nearest_neighbor_query` The input is a key `key` and a certain distance function `distance`. The query returns an iterator to the beginning of the sequence of all pairs `<k,v>` ordered by distance from `key`, where this distance is computed using the `distance` function. The distance parameter is optional; if ommited the Euclidean distance is used.

8.1.3 Iterators

Following the STL philosophy, the queries previously described return all the elements in the *k*d-tree that match the query using iterators. The iterator allows us to move thru these elements, using the increment operator `operator++()`, that moves to the next element as usual; to get the information pointed to by an iterator, the dereferencing operator `operator*()` is used.

Since each element in the *k*d-tree is an object of type `pair<Key,Value>`, when we get the element that is pointed to by an iterator we always get an object of type `pair<Key,Value>`. Then, we need to use `first` and `second` members of the STL pair to access to the key and the value, respectively.

Although there are a specific iterator defined for each query (for instance, an `orthogonal_range_query` returns an `or_iterator`), the SMTL have implemented a generic `iterator` class to facilitate end-user usage.

8.1.4 Distances

Two of the described associative queries, the `radius_range_query` and the `nearest_neighbor_query`, uses distance functions. The SMTL library includes a `Minkowski` class in "`Minkowski.hpp`" file that computes the Minkowski distance in k-dimensional spaces.

The Minkowski distance of order p, denoted L_p, is the generalized metric distance. L_1 is the city block distance or Manhattan distance; L_2 is the usual Euclidean distance. Chebyshev distance is another special case, it corresponds to L_∞.

We detail now the mathematical formula for the Minkowski distances between given points `x` and `y` and how to get it in the SMTL.

Manhattan distance:

$$dist(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

```
Minkowski<Key,1>::distance(x,y)
```

Euclidean distance:

$$dist(x, y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$

```
Minkowski<Key,2>::distance(x,y)
```

Chebyshev distance:

$$dist(x, y) = \max_i |x_i - y_i|$$

```
Minkowski<Key,-1>::distance(x,y)
```

Minkowski distance of order $p > 0$:

$$dist(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

```
Minkowski<Key,p>::distance(x,y)
```

8.1.5 Examples

In this section we present several examples of using the SMTL library in order to see the power of the library and how to work with SMTL.

To use the SMTL objects, the user has to include the file that contains the definition of the kd-tree variant that he wants to use. For instance, the `"SMTL/mdn_kdtree.hpp"` file for median kd-tree. Another simply way is to include the `"SMTL/smtl.hpp"` file that gives access to the complete library.

Creating kd-trees

We start with two programs that show how to create and insert information in a kd-tree. In the first example, we create a standard kd-tree where the type of the key and the value are a vector of integers and a string, respectively. Then, the program asks for 10 elements to insert in the kd-tree. Since the dimension is no specified during the constructor operation, it is taken from the first inserted element, and the tree has dimension two.

```
#include "SMTL/smtl.hpp"
using namespace smtl;

int main()
{
  typedef vector<int> point ;
  std_kdtree<point,string> T;

  for ( int i = 0; i < 10; ++i )
  {
    point p(2);  string value;
    cout << endl;
    cout << "x coordinate? "; cin >> p[0];
    cout << "y coordinate? "; cin >> p[1];
    cout << "value of the key? "; cin >> value;
    T.insert(p,value);
  }
}
```

The second example shows a squarish kd-tree created with the elements stored in the vector `info`, and using the constructor that receives two iterators.

```
typedef vector<int> point;

vector< pair<point,string> > info;
process_data( info );

//create the squarish kd-tree from vector
sqr_kdtree<point,string,2> T( info.begin(), info.end() );
```

Searching (`search`)

We focus now in the available associative queries. Suppose that we have a kd-tree `T`, it does not matter the variant. The elements of the kd-tree have keys which are vectors of three integers and have strings as associated values. We give an example of the `search` operation, where the key of the element to find is `<5,4,3>`. The query returns an iterator of type `srch_iterator`. If the element is in the kd-tree, the iterator points to this element, otherwise, the iterator is empty (`T.end()`). Remember that the iterator points to a `pair<Key,Value>`, so that we need to use its `first` and `second` members in order to access to the key and the value, respectively.

```
//define the integer query point
typedef vector<int> point;
point p(3); p[0] = 5; p[1] = 4; p[2] = 3;

//search in the kd-tree the point 'p'
srch_iterator<point,string> it = T.search( p );
if ( it != T.end() )
{
  cout << "The␣string␣associated␣to␣the␣given␣point␣is␣";
  cout << (*it).second << endl;
}
```

Orthogonal range (`orthogonal_range_query`)

In the following program we have a kd-tree `T` whose elements have keys which are vectors of four integers and have strings as associated values. We search for all the points that are located in a hyperrectangular region. To get them, we calls the `orthogonal_range_query` that returns an iterator that refers to the first element found, if there is such an element; moving with this iterator we can find the sequence of all the elements. To indicate the end of the sequence, it returns an empty iterator.

We support two ways to define the region: giving the two extremal corners or giving a center and the lenght of the region for each dimension. We have included an example for each one of these possibilities.

```
vector< pair<point,string> > info;

//declares the orthogonal range iterator
or_iterator<point,string> it;

//region defined by the 'lb' and 'hb' corners
point lb(4); lb[0] = 3; lb[1] = 3; lb[2] = 3; lb[3] = 3;
point hb(4); hb[0] = 7; hb[1] = 7; hb[2] = 7; hb[3] = 7;
```

```
//search and print the points in the region
it = T.orthogonal_range_query( lb, hb );
for ( it; it != T.end(); ++it )
{
      vector<int> v = (*it).first;
      cout << endl << "Point:";
      for ( int i = 0; i < v.size(); ++i )
        cout << v[i] << ",";
}

//region centered at the point 'p' with sides 'side'
point p(4); p[0] = 7; p[1] = 7; p[2] = 7; p[3] = 7;
point side(4); side[0]=4; side[1]=4; side[2]=4; side[3]=4;

//search and print the points in the region
it = T.orthogonal_range_query( p, side );
for ( it; it != T.end(); ++it )
{
      vector<int> v = (*it).first;
      cout << endl << "Point:";
      for ( int i = 0; i < v.size(); ++i )
        cout << v[i] << ",";
}
```

Partial match (`partial_match_query`)

The following example uses the `partial_match_query` to find all the elements that match in some specified coordinates with a given key. In this case, the data structure is a `rlx_kdtree` whose keys are of type `Employee`. In this example, the data is stored in the key, and the value is defined as a string but contain no information.

```
class Employee{
public:
  string name;
  string city;
  int    age;
  string degree;
}
```

In order to apply the SMTL queries, the `Employee` class has to support the following members:

- `size()`: returns the number of fields of Employee.
- `operator[](int i)`: returns the `i`'th field.
- `operator==(Employee e)`: returns true when the two `Employee` objects are equal.

We insert five elements in the kd-tree and we define an `Employee e` that will be the key of the query. `coords` is the vector where are specified which coordinates has to match the values of the query key. In the first query, we specify the degree, and in the second one, we specify the city and the degree. The query returns an `pm_iterator` iterator with all the keys stored in the data structure that match these coordinates. Then, moving through the iterator we can get all the elements that match the specified query.

```
int main()
{
  //create the kdtree and insert Employees
  rlx_kdtree<Employee,string> T;

  T.insert( Employee("Peter","Paris",29,"Maths"), "");
  T.insert( Employee("John","London",53,"Maths"), "");
  T.insert( Employee("Anna","London",45,"Physics"), "");
  T.insert( Employee("Bill","Paris",34,"Physics"), "");
  T.insert( Employee("Maria","Paris",25,"Maths"), "");

  //define the query Employee for the first query
  Employee e1 ("", "", 0, "Maths");

  //define the vector of specified coordinates
  //only the 'degree' attribute is relevant
  vector<bool> coords(e1.size());
  coords[0] = false; coords[1] = false;
  coords[2] = false; coords[3] = true;

  //declare the partial match iterator
  pm_iterator<Employee,string> it;

  //run the first query
  cout << "Degree␣in␣Maths:" << endl;
  it = T.partial_match_query( e1, coords );
  for ( it; it != T.end(); ++it )
    cout << "Name:" << (*it).first.name << endl;

  //define the query Employee for the second query
  Employee e2 ("", "Paris", 0, "Maths");

  //define the vector of specified coordinates
  //only the 'city' and the 'degree' attribute are relevant
  coords[0] = false; coords[1] = true;
  coords[2] = false; coords[3] = true;

  //run the second query
  cout << "Degree␣in␣Maths,␣lives␣in␣Paris:" << endl;
  it = T.partial_match_query( e2, coords );
  for ( it; it != T.end(); ++it )
    cout << "Name:" << (*it).first.name << endl;
}
```

The output from the above example looks like this:

```
Degree in Maths:
Name: Peter
Name: Maria
Name: John

Degree in Maths, lives in Paris:
Name: Peter
Name: Maria
```

Linear match (`linear_match_query`)

Next program calls the `linear_match_query` to search in the kd-tree all the points that are located on a given line. It returns an iterator that refers to the first element found, if there is such an element; moving with this iterator we can find the sequence of all the elements stored in the kd-tree that are located on that line. To indicate the end of the sequence, it returns an empty iterator.

This operation can be used defining the line with two points or with a point and the slope. For this reason, we support two versions of the query. Note that this operation only makes sense in a two dimensional space with numerical keys.

```
typedef vector<int> point;

//declare the linear match iterator
lm_iterator<point,string> it;

//line defined by the 'p1' and 'p2' points
point p1(2); p1[0] = 5; p1[1] = 4;
point p2(2); p2[0] = 4; p2[1] = 0;

//run the linear match query and print the points on the line
it = T.linear_match_query( p1, p2 );
for ( it; it != T.end(); ++it )
{
  vector<int> v = (*it).first;
  cout << endl << "Point:";
  for ( int i = 0; i < v.size(); ++i )
    cout << v[i] << ",";
}

//line defined by the 'p' point and the slope 's'
point p(2); p[0] = 3; p[1] = 1;
int s = 3;

//run the linear match query and print the points on the line
it = T.linear_match_query( p, s );
```

```
for ( it; it != T.end(); ++it )
{
  vector<int> v = (*it).first;
  cout << endl << "Point:";
  for ( int i = 0; i < v.size(); ++i )
    cout << v[i] << ",";
}
```

Radius range (`radius_range_query`)

Suppose now that we have a kd-tree that stores information about restaurants and their menu price. The key of an element is the geographical location of the restaurant and the value is an integer with the price of the menu. We want the list of restaurants that are inside a specific radius from a given point. Moreover, we want that the distance function used is the Manhattan distance. The query that allows us to get this information is the `radius_range_query`, that returns via an iterator all the elements in the kd-tree that satisfies the distance constraint.

```
//define the given location and the maximum distance
typedef vector<int> point;
point c(2); c[0] = 4; c[1] = 2;
int dist = 5;

//search the restaurants in a radius 'dist' of point 'p'
//list the price and the location of the restaurants
rr_iterator<point,int> it;
it = T.radius_range_query(c,dist,Minkowski<point,1>::distance);
for ( it; it != T.end(); ++it )
{
  cout << endl <<  "Price:" << (*it).second;
  int d = Minkowski<point,1>::distance(c,(*it).first);
  cout << "␣␣distance:" << d;
}
```

The output of the program is the price and the location of all the restaurants that are, at most, to a Manhattan distance 5 of the given location. For instance, the output could be the following.

```
Price:9  distance:4
Price:8  distance:2
Price:19  distance:5
Price:18  distance:4
```

The previous program returns the restaurants as it found them, without any specific order. Suppose that we want the restaurants listed in increasing order of the price of the menu. An easy way to get this information is combining the result of the `radius_range_query` with STL operations. First

of all we need to define an operation `smallerPrice(x,y)` that compares two restaurants, `x` and `y`, and returns true when the first one has a menu price smaller than the second one.

```
class smallerPrice{
public:
  bool operator()(pair<point,int> x, pair<point,int> y) const{
        return x.second < y.second;
  }
};
```

The program runs the `radius_range_query` and moves the elements returned by the query to a vector, using the `copy` function. This function requires two iterators of the same type, so that the generic iterators are used instead of the query specific one. Finally, it applies the STL `sort` algorithm with the `smallerPrice(x,y)` predicate to sort the vector by the menu price.

```
//define the given location and the maximum distance
typedef vector<int> point;
point c(2); c[0] = 4; c[1] = 2;
int dist = 5;

//define the generic iterators to use in the STL algorithm
iterator<point,int> it;
it = T.radius_range_query(c,dist,Minkowski<point,1>::distance);
iterator<point,int> it_end = T.end( );

//create a vector with the elements found in the area
vector< pair<point,int> > elems;
copy( it, it_end, back_inserter(elems) );

//sort this vector using the smallerPrice function
sort( elems.begin(), elems.end(), smallerPrice() );

//list the price and the location of the restaurants
vector< pair<point,int> >::const_iterator it_elem;
for ( it_elem=elems.begin(); it_elem != elems.end(); ++it_res )
{
  cout << endl <<  "Price:" << (*it_elem).second;
  int d = Minkowski<point,1>::distance(c,(*it).first);
  cout << "␣␣distance:" << d;
}
```

Comparing this output and the previous one, we can see that the content is exactly the same, but in the second version the restaurants are listed in increasing order by the menu price.

```
Price:8  distance:2
Price:9  distance:4
Price:18  distance:4
Price:19  distance:5
```

Nearest neighbor(`nearest_neighbor_query`)

The following example explores information about the stores of our city that are held in the kd-tree. The kind of the store is the value of the element (for instance, electronics, pharmacy, clothes...) and its geographical location is the key.

Suppose that we want to recover the nearest pharmacy to one specified location. The suitable associative query that answers this question is the `nearest_neighbor_query`, giving the location as a parameter. Another optional parameter is the `distance` function used in the algorithm, that in this case it is not specified. When the query call does not define any distance function, the algorithm takes the Euclidean distance as default. The program runs the query to obtain an iterator with the stores in increasing distance order. Then, it traverses sequence using the iterator and checking if the store is a pharmacy until it gets the first or no pharmacy if found.

```
//define the given location
typedef vector<int> point;
point p(2); p[0] = 3; p[1] = 1;

//run the incremental nearest neighbor search
nn_iterator<point,string> it = T.nearest_neighbor_query( p );
bool found = false;
for ( it; !found && it != T.end(); ++it )
{
  //check if the element found is a pharmacy
  if ( (*it).second == "Pharmacy" )
  {
    cout << "The␣nearest␣pharmacy␣is␣in␣location␣";
    vector<int> v = (*it).first;
    for ( int i = 0; i < v.size(); ++i )
      cout << v[i] << ",";
    found = true;
  }
}
```

We propose another solution to the previous example using the `find_if` STL algorithm. This algorithm requires an iterator to explore and the predicate to be satisfied. It searches a sequence for the first occurrence of an element for which the given predicate is true. In our program, we need to define an unary predicate object type, `IsPharmacy()`, that returns true when the values associated to a key is `Pharmacy`.

```
class IsPharmacy{
public:
  bool operator()(pair<point,string> x) const{
        return x.second == "Pharmacy";
  }
};
```

As we have said in the previous example, the STL algorithms needs a `begin()` and `end()` iterators of the same type. Then, we can not use the SMTL query iterators and we have to convert them into the generic one. After that, we run the `find_if` algorithm and the returned iterator points to the nearest pharmacy to the specified location or to an empty iterator if there is no pharmacy.

```
//define the given location
typedef vector<int> point;
point p(2); p[0] = 3; p[1] = 1;

//define the generic iterators to use in the STL algorithm
iterator<point,string> it1 = T.nearest_neighbor_query(p);
iterator<point,string> it_end = T.end();

//run the STL find_if using SMTL iterators
iterator<point,string> it2 = find_if(it1,it_end,IsPharmacy());
if ( it2 != it_end )
{
  //the iterator points to the nearest pharmacy
  cout << "The␣nearest␣pharmacy␣is␣in␣location␣";
  vector<int> v = (*it2).first;
  for ( int i = 0; i < v.size(); ++i )
        cout << v[i] << ",";
}
```

Generic iterators

The last example shows a program where only the generic iterators are used. The program declares an `iterator it` and assigns to it the result of several queries.

```
//declares the generic iterators
iterator<point,int> it;
iterator<point,int> end = T.end();

//runs a search query
point p(2); p[0] = 7; p[1] = 7;
it = T.search( p );
if ( it != end )
  cout << "Search:␣element␣found" << endl;


//runs a linear match query
point p2(2); p2[0] = 5; p2[1] = 4;
it = T.linear_match_query( p, p2 );
int k = 0;
for ( it; it != end; ++it )
  k++;
cout << "Linear␣match:␣found␣" << k << "␣elements" << endl;
```

```
//runs a radius range query
k = 0;
it = T.radius_range_query( p, 5 );
for ( it; it != end; ++it )
  k++;
cout << "Radius␣range:␣found␣" << k << "␣elements" << endl;
```

The output is,

```
Search: element found
Linear match: found 5 elements
Radius range: found 35 elements
```

8.2 Implementation details

We have mentioned previously that the queries do not compute the sequence of elements to return in advance, and that the iterators always searches the next element when the user asks for it. Since the recursive algorithms presented in Chapter 2 do not allow us to do that, we have implemented in the STML library an incremental iterative version. This iterative version is more efficient because it saves the recursive calls, and it only searches for the number of elements that the user needs.

In this section we only detail the `orthogonal_range_query` as example; the others queries are implemented in a similar way. The algorithm is implemented in the `or_iterator` class ("`or_iterator.hpp`" file). This iterator stores the lowerBound and the upperBound of the region as a members, `lb` and `ub`, respectively. Moreover, it needs a `queue` with the nodes that has to be visited in the future.

The kd-tree calls the `init` member giving as a parameter the root of the tree to analyze; the operation `or_search()` searches for the next element located inside the defined range, and stops when this element is located on the top of the queue or when the queue is empty (means that there are not more elements in the region). If the queue is not empty, the `operator*()` only checks the top of the queue in order to get the element and return it; and to increment the iterator, the `operator++()` removes the top of the queue and call `or_search()` again, that looks for the next element.

```
template <typename Key, typename Value, unsigned int Dim=0>
class or_iterator
{
private:
  queue<typename kdtree<Key,Value,Dim>::node*>  q;
  Key lb;  //lowerBound
  Key ub;  //upperBound
```

```
public:
  //initializes the query
  //operation called by the kdtree orthogonal_range_query
  //the element 'n' is always the root of the tree to analyze
  void or_iterator<Key,Value,Dim>::init(
                    typename kdtree<Key,Value,Dim>::node* n )
  {
    q.push( n );
  }


  //increment operator
  or_iterator<Key,Value,Dim>&
  or_iterator<Key,Value,Dim>::operator++()
  {
    if ( q.size() > 0 )  //there are nodes to visit
    {
      q.pop();       //removes the current element
      or_search();   //looks for the next one
    }
    return *this;
  }


  //dereference operator
  //gets the element on the top of the queue and
  //returns the key and the value stored there
  pair<Key,Value>
  or_iterator<Key,Value,Dim>::operator*() const
  {
    typename kdtree<Key,Value,Dim>::node* n = q.front();
    return make_pair( n->key, n->value );
  }


  //orthogonal range search algorithm
  void or_iterator<Key,Value,Dim>::or_search()
  {
    bool found = false;
    while ( !found && !q.empty() )
    { //visit nodes until it gets one element located in the
      //region or there are no more elements to visit

      //get the key and the discriminant on the top of the queue
      typename kdtree<Key,Value,Dim>::node* n = q.front();
      Key key = n->key;
      unsigned int discr = n->discr;

      if ( n->left != NULL && lb[discr] < key[discr] )
        //left subtree bounding box intersects the region
        q.push( n->left );
      if ( n->right != NULL && key[discr] <= ub[discr] )
        //right subtree bounding box intersects the region
```

```
        q.push( n->right );

      //checks if the element is located in the region
      bool inRange = true;
      for ( unsigned int i = 0; inRange && i < lb.size(); ++i )
          inRange = (lb[i]<=key[i]) && (key[i]<=hb[i]);

      if ( inRange )  //element in the region
        found = true;  //stop in the next iteration
      else  //element not in the region.
        q.pop();  //remove the top of the queue and continue
    }
  }
};
```

Regarding the `nearest_neighbor_search`, the incremental algorithm has some differences from the algorithm that returns only the first nearest neighbor element (presented in Section 2.7).

In the incremental version, when the algorithm analyzes an element, it stores in the priority queue (sorted by increasing order of distance) the *real* distance between this element and the given point. Moreover it stores the *potential* distance for their left and right subtrees, defined as the minimum possible distance between a point lying inside the bounding box of those subtrees and the given point. Then, the algorithm iterates until an element with real distance reaches the top of the priority queue. This means that this element is the next the nearest neighbor: the algorithm has not found elements with distance smaller than the current one and there are not subtrees in the queue with smaller potential distance either. The `nodeInfo` structure stores the node, its bounding box, the distance to the given point and a bit to indicate if its computed distance is real or potential.

```
template <typename Key, typename Value, unsigned int Dim>
void nn_iterator<Key,Value,Dim>::nn_search()
{
  if ( pq.empty() )  //all the elements have been visited
    return;

  while ( !pq.top().real )
  { //the top of the priority queue

    //get information about the element on the top of the queue
    nodeInfo nb = pq.top();
    typename kdtree<Key,Value,Dim>::node* n = nb.n;
    Key key = n->key;
    unsigned int discr = n->discr;

    pq.pop();  //remove the current top of the priority queue

    //compute the real distance and push in the queue again
    nodeInfo x = nodeInfo( n, distance(center, key), true );
```

```
    pq.push( x );

    if ( n->left != NULL ) {
      //compute the bounding box for the left subtree
      Key maxBound = nb.maxBound;
      maxBound[discr] = key[discr];

      //compute the potential distance and push the left subtree
      double dist = minimun_distance_box(nb.minBound,maxBound);
      nodeInfo x1 = nodeInfo( n->left, dist, false,
                                        nb.minBound,maxBound );
      pq.push( x1 );
    }

    if ( n->right != NULL ) {
      //compute the bounding box for the right subtree
      Key minBound = nb.minBound;
      minBound[discr] = key[discr];

      //compute the potential distance and push right subtree
      double dist = minimun_distance_box(minBound,nb.maxBound);
      nodeInfo x1 = nodeInfo( n->right, dist, false,
                                       minBound, nb.maxBound );
      pq.push( x1 );
    }
  }
};
```

Chapter 9

Conclusions

This master thesis has been focused on a well-known data structure for storing multidimensional points: the kd-tree. As we have shown, kd-trees are relatively easy to understand and implement, and they are also useful for several important queries involving multidimensional keys, like orthogonal range searches, partial match queries and nearest neighbor searches, among others.

In Chapters 2 through 4 we have recalled some of the kd-tree variants in the literature: the *standard kd-tree* of Bentley [Ben75], the *squarish kd-tree* of Devroye, Jabbour and Zamora-Cura [DJZC00] and the *relaxed kd-tree* of Duch, Estivill-Castro and Martínez [DECM98].

For the standard kd-tree variant, we have included some previously known average cost analyses: those for a full-defined search and for a partial match operation. By contrast, the analysis presented for the expected cost of a linear match query is our first contribution. We have carried out this analysis exclusively for standard kd-trees, and in one of two given models, for two main reasons. On the one hand, the experiments under that model for this operation suggest that for other variants of kd-trees the cost does not depend on the slope of the query line. On the other hand, computing the expected cost of the linear match operation under the other model and for the other kd-tree variants seems a very difficult task, which we left as future work.

In Chapter 5 we have proposed a new kd-tree variant, which we have named *median kd-tree*. These kd-trees are on the average more balanced than the rest of kd-trees, since they take into account the point that is inserted at each moment so as to always choose as discriminant the dimension that cuts the region more evenly. Furthermore, when the number of dimensions of the space increases, the expected cost of a search tends to $\log_2 n$,

which is optimal. Unfortunately, the expected costs of the other operations considered in this work are larger for median kd-trees than for standard kd-trees. We have included in this chapter the theoretical analysis of the asymptotic cost of a full-defined random search for any value of k, and also of the cost of the partial match operation for $k = 2$ and $k = 3$. The analysis for partial matches can be extended for any $k \geq 4$ following exactly the same steps, but the calculations are a bit cumbersome. This analysis is also left as future work.

The three hybrid data-structures that we propose in Chapter 6 are another contribution of this master thesis. These data structures combine standard kd-trees with the squarish, median and relaxed variants, thus obtaining *hybrid squarish*, *hybrid median* and *hybrid relaxed* kd-trees. Perhaps the most remarkable of these data structures is the hybrid median kd-tree, which produces both more balanced trees and more squarish regions than the standard variant. For this reason, its expected search cost is only improved by the median kd-tree, and its expected cost for the partial match is only improved by the squarish kd-tree. Additionally, we have shown experimentally that, as it is intuitively expected, the cost of the search improves as the number of dimensions of the space grows. This improvement with the number of dimensions is similar to (but not as fast as) that of the median kd-tree. In particular, the expected cost of the search also seems to tend to $\log_2 n$ as k tends to infinity.

We have also carried out an exhaustive experimental work for all the associative queries and for the seven kd-tree variants of our interest, in some cases for dimensions larger than 2. The results of the experiments are presented in Chapter 7. All those results completely match with the theoretical results presented along this master thesis.

Moreover, we have designed and developed an efficient and generic library of metric and spatial data structures and algorithms that follows the philosophy of the C++ *Standard Template Library*, or STL for short. We have named this new library *Spatial and Metric Template Library*, or SMTL for short. The SMTL implements all associative queries presented in Chapter 2, for the seven kd-tree variants of this master thesis. The SMTL is robust, flexible and has been thoroughly tested. It provides ready-to-use solutions to common queries when dealing with multidimensional data, like nearest neighbor searches or orthogonal range searches. As an implementation design, it was decided to write this library so to use an iterator-like interface to return the results of the queries. This approach allows the user to write code that combines the STL and the SMTL in an easy and convenient way. Of course, the library can be extended and improved in several aspects, most notably by including some metric containers like *Vantage Point* trees or *Burkhard-Keller* trees [CNByM99]. These containers only support

nearest neighbor and radius range queries as they only work with a distance function; the objects stored in such a containers need not to have attributes or coordinates or a fixed dimensionality.

To finish, Table 9.1 below summarizes the expected cost for a full-defined search and for a partial match for the seven variants of kd-trees considered in this work.

kd-tree variant	expected cost			
	search $c \log_2 n$		partial match n^α	
	$k = 2$ c	$k \to \infty$ c	$k = 2$ α	
standard	1.38628...	1.38628...	0.56155...	[Knu98][FP86]
squarish	1.38628...	1.38628...	0.5	[DJZC00]
median	1.15086...	1	0.60196...	§5.1
relaxed	1.38628...	1.38628...	0.61803...	[DECM98]
hybrid squarish	1.38628...	1.38628...	$> 0.54595...$[1]	§6.2.1
hybrid median	1.25766...	1	0.54595...	§6.1.1
hybrid relaxed	1.38628...	1.38628...	0.56155...	§6.3.1

Table 9.1: Expected costs of the seven kd-tree variants.

[1] Experimentally

List of Figures

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[CDZc99] Philippe Chanzy, Luc Devroye, and Carlos Zamora-cura. Analysis of range search for random k-d trees. *Acta Informatica*, 37:355–383, 1999.

[CNByM99] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-yates, and José L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33:273–321, 1999.

[DECM98] Amalia Duch, Vladimir Estivill-Castro, and Conrado Martínez. Randomized k-dimensional binary search trees. In K.-Y. Chwa and O.H. Ibarra, editors, *Proc. of the 9nth Int. Symp. on Algorithms and Computation (ISAAC)*, volume 1533 of *Lecture Notes in Computer Science*, pages 199–208. Springer-Verlag, 1998.

[DJZC00] Luc Devroye, Jean Jabbour, and Carlos Zamora-Cura. Squarish k-d trees. *SIAM Journal on Computing*, 30:1678–1700, 2000.

[DM02] Amalia Duch and Conrado Martínez. On the average performance of orthogonal range search in multidimensional data structures. *J. Algorithms*, 44:226–245, 2002.

[Duc04] Amalia Duch. *Design and analysis of multidimensional data structures*. PhD thesis, Universitat Politècnica de Catalunya, 2004. Advisor: Conrado Martínez.

[FNC08] Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. *SISAP Library*, 2008.

[FP86] Philippe Flajolet and Claude Puech. Partial match retrieval of multidimensional data. *J. ACM*, 33(2):371–407, 1986.

[Knu98] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[Mou10] David M. Mount. *ANN Programming Manual*, 2010.

[Rou01] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *J. ACM*, 48(2):170–205, 2001.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.